Host Integration Server 2000

Microsoft Host Integration Server 2000

Microsoft Host Integration Server provides comprehensive host access and integration, extending Microsoft Windows to other systems by integrating host applications, data sources, messaging, and security systems. This enables the reuse of IBM mainframe and midrange data and applications across distributed environments.

Host Integration Server offers a strong, dependable, and secure platform on which to leverage and extend existing investments in SNA and Web technologies. Host Integration Server enables rapid adaptation to new business opportunities while preserving existing infrastructure investments.

In This Library Section	Essentials
• **Microsoft Host Integration Server 2000 Developer's Guide** • **Microsoft Host Integration Server 2000 Product Overview** • **Technical Articles**	• **Product Documentation (Books Online)** • **IT Resources** • **Additional Product Documentation** • **Frequently Asked Questions** • **Host Integration Server Community** • **Newsgroup**

Microsoft Host Integration Server 2000 Developer's Guide

Microsoft® Host Integration Server 2000 provides comprehensive bi-directional services for integrating Microsoft Windows® with legacy systems. Host Integration Server 2000 extends Windows to other platforms by providing interoperability in three areas:

- Application Integration Services
- Data Integration Services
- Network Integration Services

For more information on the features and use of Host Integration Server 2000, see the topics in the Getting Started section of the on-line help.

Most of the services provided by Host Integration Server 2000 expose a programming interface which allows you to extend the functionality of the product and integrate it more tightly in your own environment. This guide describes these interfaces, and provides guidance on how to use them.

This section contains:

Application Integration

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information required to develop software to integrate applications in an environment using Microsoft Host Integration Server 2000.

This section contains:

Introduction to Application Integration

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop software to integrate applications in an environment using Microsoft® Host Integration Server 2000. Several methods are provided using Host Integration Server for integrating applications:

- COM Transaction Integrator (COMTI) to integrate COM applications with CICS and IMS transactions on IBM mainframes and minicomputers (see the COM Transaction Integrator for CICS and IMS section under Application Integration Services in the Host Integration Server user documentation).
- MSMQ-MQSeries Bridge to develop applications to send messages between IBM MQSeries and Microsoft Message Queue Server (MSMQ) in an environment using the Microsoft MSMQ-MQSeries Bridge in Microsoft Host Integration Server 2000.
- Data access and data tools to develop applications for data access, data replication, and data tools for integrating with AS/400 and VSAM files on IBM mainframes and minicomputers and IBM DB2 databases hosted on IBM VMS, OS/390, AS/400, IBM AIX, Microsoft Windows® 2000, and Microsoft Windows NT® Server (see Introduction to Data Integration).

COMTI allows developers to integrate mainframe-based transaction programs (TPs) with component-based Windows applications. With COMTI, you can integrate existing mainframe-base transaction programs with Windows-based COM or distributed COM (DCOM) applications. You may not even have to modify your mainframe TP if the business logic is separate from the presentation logic. The wizards available in the COMTI Component Builder and COMTI Manager guide you through the process, step-by-step.

COMTI is appropriate when a synchronous or transactional solution is needed where both systems being integrated are running at all times. For applications only requiring an asynchronous integration solution, a messaging-based solution using the MSMQ-MQSeries Bridge is preferred over COMTI.

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop applications to send messages between IBM MQSeries and Microsoft Message Queue Server (MSMQ) in an environment using the Microsoft MSMQ-MQSeries Bridge in Microsoft Host Integration Server 2000. This section provides documentation for developers on programming issues using the MSMQ-MQSeries Bridge including message queue naming, message conversion, and the MSMQ-MQSeries Bridge Extensions.

Applications that integrate message queuing and the use MSMQ-MQSeries Bridge in a Host Integration Server 2000 environment can be developed using several different development tools and application programming interfaces including:

- C or C++ applications that use the MSMQ-MQSeries Bridge Extensions to extend the MSMQ-MQSeries Bridge.
- Microsoft Visual Basic® applications that use MSMQ-MQSeries Bridge Extensions to extend the MSMQ-MQSeries Bridge.

To use this guide effectively, you should be familiar with:

- Microsoft Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows 2000
    - Microsoft Windows NT Server or Workstation
    - Microsoft Windows 98
    - Microsoft Windows 95
- Microsoft Message Queue Server
- IBM MQSeries

Depending on the application programming interface and development tools used, you should be familiar with:

- Microsoft COM objects
- Active Server Pages

This section contains:

- Additional Resources

Additional Resources

This guide does not describe the products, architectures, or standards developed by other companies or organizations. For information about Microsoft Windows 2000, Microsoft Windows NT Server, and other operating systems, consult your product documentation.

For information about SNA architecture, refer to your system network documentation.

The following documents provide additional information about the Microsoft Message Queue Server:

- *Message Queuing (MSMQ)* in the Microsoft Developer Network (MSDN®) Platform Software Development Kit

For more information about SNA see the following manuals:

- *IBM Systems Network Architecture: Technical Overview*
- *IBM Systems Network Architecture: Concepts and Products*
- *IBM SNA Format and Protocol Reference Manual: Architectural Logic*

For more information about IBM MQSeries, see the following manuals:

- *IBM MQSeries: An Introduction to Messaging and Queuing* (Document Number GC33-0805)
- *IBM MQSeries Intercommunication* (Document Number SC33-1872)
- *IIBM MQSeries Messages* (Document Number GC33-1876)

For background information about logical unit (LU) 6.2, Advanced Program-to-Program Communications (APPC), or the Common Programming Interface for Communications (CPI-C), see the following manuals:

- *IBM SNA: Technical Overview*
- *IBM SNA: Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM SNA: Formats*
- *IBM Systems Network Architecture: Introduction to APPC*
- *IBM Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2*

MSMQ-MQSeries Bridge Programming

The MSMQ-MQSeries Bridge provides the ability to send and receive messages between Microsoft Message Queue Server (MSMQ) and IBM MQSeries and easily and efficiently. The MSMQ-MQSeries Bridge Extensions allow a programmer to develop and control how these message transfers will occur and how properties of a message are translated.

The main programming issues when using the MSMQ-MQSeries Bridge Extensions fall into three areas:

- Queue addressing
- Message conversion
- Limitations to specific API functions

Queue addressing deals with how to specify the name of an MQSeries destination queue in an MSMQ API call, or the name of an MSMQ destination queue in an MQSeries API call. Message conversion deals with how the MSMQ-MQSeries Bridge converts MSMQ message properties to MQSeries message data structures, and how the MSMQ-MQSeries Bridge converts MQSeries message data structures to MSMQ properties. These topics are covered in detail in separate sections below. Limitations to the MSMQ-MQSeries Bridge Extensions are discussed in detail in the section on programming considerations.

This section contains:

Platforms Supported by MSMQ-MQSeries Bridge Extensions

The Microsoft MSMQ-MQSeries Bridge Extensions can access message queues on the following IBM MQSeries systems through SNA LU6.2 or TCP/IP using Microsoft® Host Integration Server 2000 and the MSMQ-MQSeries Bridge:

- IBM MQSeries for OS/390 Version 2 Release 1 (V2.1)
- IBM MQSeries for AS/400 Version 4 Release 3 (V4R3MO)
- IBM MQSeries for Windows NT Version 5 Release 1 (V5.1)
- IBM MQSeries for Windows NT Version 5 (V5.0)
- IBM MQSeries for Windows NT Version 2 (V2.0)

The Microsoft MSMQ-MQSeries Bridge Extensions require the following computer-to-host connectivity software when using SNA LU6.2 as the network transport:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server 2000 End User Client
- Microsoft Host Integration Server 2000 Administrator Client

The Microsoft MSMQ-MQSeries Bridge Extensions require that the Microsoft MSMQ-MQSeries Bridge be installed on one of the following operating systems:

- Microsoft Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Data Center
- Microsoft Windows NT® Server 4.0 Enterprise Edition with Service Pack 6a or later

On Windows 2000, the MSMQ-MQSeries Bridge requires that Message Queuing be set up as an MSMQ server, not a workgroup, with routing enabled. On Windows NT 4.0 Enterprise Edition, the prerequisites require MSMQ 1.0 be installed (MSMQ can be installed from the Windows NT 4.0 Option Pack) and MSMQ be set up as a Routing Server, Primary Enterprise Controller (PEC), Primary Site Controller (PSC) or Backup Site Controller (BSC).

The Microsoft MSMQ-MQSeries Bridge Manager used to configure and manage the MSMQ-MQSeries Bridge can be installed on one of the following operating systems:

- Microsoft Windows 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Data Center
- Microsoft Windows 2000 Professional
- Microsoft Windows 2000 with Terminal Service
- Microsoft Windows NT Server 4.0 Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0 Terminal Server Edition with Service Pack 6 or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later

On Windows 2000 Professional or Windows 2000 with Terminal Service, the MSMQ-MQSeries Bridge Manager requires that Message Queuing not be set up in a workgroup. On Windows NT Workstation 4.0 or Windows NT Server 4.0 Terminal Server Edition, any configuration of MSMQ 1.0 is supported.

The MSMQ-MQSeries Bridge supplied with Host Integration Server 2000 supports only the Windows 2000 and the Intel Windows NT 4.0 Enterprise Edition platforms. Older versions of SNA Server 4.0 supported Windows NT on the Alpha architecture, however this configuration is not supported by the MSMQ-MQSeries Bridge.

IBM MQSeries support is available on a variety of other platforms. The MQSeries-MSMQ Bridge has not been tested with these other implementations.

Queue Addressing Using MSMQ-MQSeries Bridge

This section explains how to specify the name of an MQSeries destination queue in an MSMQ call, or the name of an MSMQ destination queue in an MQSeries call. This information is needed to specify the destination queue where you are sending a message.

This section contains:

Addressing an MQSeries Queue in MSMQ

In MSMQ, address MQSeries queues as if they are on the MSMQ network. For the MSMQ machine name, specify the MQSeries Queue Manager name. For the MSMQ queue name specify the MQSeries queue name.

For example, if the MQSeries Queue Manager is **MQS1** and the queue name is **MQS_QUEUE4**, then the MSMQ path name is **MQS1\MQS_QUEUE4**.

Note For detailed information on queue addressing, see the section on Queue format names under Converting Messages from MSMQ to MQSeries

Sending a Message to an MQSeries Queue in MSMQ

To send a message to an MQSeries queue, follow the normal MSMQ procedure. Determine the MSMQ format name corresponding to the path name. More precisely, you must determine the format name of the MSMQ foreign queue representing the MQSeries queue. If the foreign queue does not already exist, you can create it and determine its format name by calling **MQCreateQueue**. If the foreign queue already exists, then you can call **MQPathNameToFormatName**, or you can determine the format name in the MSMQ Explorer.

Call **MQOpenQueue** with the format name argument to open the queue for send access.

Call **MQSendMessage** and specify the destination queue handle returned by **MQOpenQueue**.

In the MSMQ-to-MQSeries direction, MSMQ-MQSeries Bridge sends transacted messages using the MSMQ->MQS message pipe and untransacted messages using the MSMQ->MQS Transactional message pipe.

Addressing an MSMQ Queue in MQSeries

There are two ways to address an MSMQ queue from MQSeries:

- By the MSMQ format name
- By the MSMQ path name

By either method, you specify the name in the object descriptor (**MQOD** structure) of the MQSeries message.

Note MSMQ-MQSeries Bridge supports the following types of format names:

```
PUBLIC=<GUID>
PRIVATE=<machine GUID>\<file number>
DIRECT=OS:<Path name>
```

Converting Messages Sent from MQSeries to MSMQ

Sending a Message to an MSMQ Queue in MQSeries

To send a message from MQSeries to MSMQ, use the following procedure. Specify the MSMQ format or path name of the destination queue in the object descriptor. Call **MQOPEN** to open the queue. Call **MQPUT** or **MQPUT1** to send the message.

The addressing syntax lets you send a message by either MQS->MSMQ message pipe or MQS->MSMQ Transactional message pipe. For an explanation of the service types, see the section on Normal and High Service.

Converting Messages Using MSMQ-MQSeries Bridge

This section describes how the MSMQ-MQSeries Bridge converts MSMQ message properties to MQSeries message data structures and how MQSeries data structures are converted to MSMQ message properties.

When the MSMQ-MQSeries Bridge transmits a message, it converts the message properties between the MSMQ and MQSeries formats. When equivalent properties exist in the two systems, the MSMQ-MQSeries Bridge assigns the property values directly. For example, the MSMQ *messagebody* property is converted to the MQSeries *messagebuffer*. The MSMQ *messagebodylength* is converted to the MQSeries *messagebufferlength*.

When partially equivalent properties exist in the two systems, the MSMQ-MQSeries Bridge assigns the properties according to conversion rules. For example, the MQSeries property **MQMD.Report** is converted to the MSMQ properties **PROPID_M_ACKNOWLEDGE** and **PROPID_M_JOURNAL**.

When a property has no equivalent, the MSMQ-MQSeries Bridge either ignores the property or assigns a default value. For example, the MSMQ property **PROPID_M_AUTH_LEVEL** refers to a specific MSMQ authentication method that is not supported by MQSeries. In a message sent from MSMQ to MQSeries, this property is ignored. In a message received by MSMQ from MQSeries, the MSMQ-MQSeries Bridge assigns the MSMQ default value to the property.

You can supplement or override the conversions described here using the MSMQ message extension property (**PROPID_M_EXTENSION**). For information on overriding the default conversions, see Message Extensions.

When you send a message from MSMQ (Microsoft Message Queue) to IBM MQSeries, MSMQ-MQSeries Bridge converts the MSMQ message properties to an MQSeries data structure. In order to do this, MSMQ-MQSeries Bridge maps the various message properties of the MSMQ message as nearly as possible to equivalent MQSeries fields. The following sections describe the conversion rules by which this is done.

This section contains:

Converting Messages Sent from MSMQ to MQSeries

The information in this section applies to messages that you send from MSMQ to MQSeries. For messages sent from MQSeries to MSMQ, see Converting Messages from MQSeries to MSMQ.

This section is divided into two main subsections, which provide essentially the same information but from complementary points of view.

The first section on Converting MSMQ Properties describes the conversion rules from the sender's point of view. This section explains how MSMQ-MQSeries Bridge converts each MSMQ property that you include in a message.

The second section on Building an MQSeries Message explains the rules from the receiver's point of view. Read this section to learn how MSMQ-MQSeries Bridge builds a complete MQSeries message containing all the needed fields, whether or not they have exact MSMQ equivalents.

You can supplement or override the conversions described here using the MSMQ message extension property (**PROPID_M_EXTENSION**).

This section contains:

Converting MSMQ Properties

This section explains how MSMQ-MQSeries Bridge converts the MSMQ properties that you include in a message to MQSeries. For information on MQSeries fields that have no MSMQ equivalents, see the following section on Building an MQSeries Message.

Message Body (PROPID_M_BODY)

The MSMQ message body is equivalent to the **MQPUT** or **MQGET** message buffer of MQSeries. The length of the **MQPUT** buffer is the MSMQ message body size.

The following table lists the MSMQ properties and the MQSeries fields to which they are converted:

MSMQ Property	Converted to MQSeries Field
PROPID_M_BODY	Message buffer
PROPID_M_BODY_SIZE	Message buffer length

Queue Format Names (PROPID_M_..._QUEUE)

Each MSMQ message contains the format name of a destination queue, and optionally the format names of response and/or administration queues. The MSMQ-MQSeries Bridge converts the names to the equivalent MQSeries object and queue manager names.

If a message contains both a response queue and an administration queue, MSMQ-MQSeries Bridge ignores the administration queue.

The following table lists the MSMQ properties and the MQSeries fields to which they are converted:

MSMQ Properties	Converted to MQSeries Fields
PROPID_M_DEST_QUEUE and PROPID_M_DEST_QUEUE_LEN	MQOD.ObjectName and MQOD.ObjectQMgrName
PROPID_M_RESP_QUEUE and PROPID_M_RESP_QUEUE_LEN	MQMD.ReplyToQ and MQMD.ReplyToQMgr
PROPID_M_ADMIN_QUEUE and PROPID_M_ADMIN_QUEUE_LEN	MQMD.ReplyToQ and MQMD.ReplyToQMgr

Message Class (PROPID_M_CLASS)

MSMQ-MQSeries Bridge converts the MSMQ message class (**PROPID_M_CLASS**) to the MQSeries message type and feedback. It translates the message class values according to the following table.

Value of MSMQ property	Converted to MQSeries value	
PROPID_M_CLASS	MQMD.MsgType	MQMD.Feedback
MQMSG_CLASS_NORMAL	MQMT_REQUEST or MQMT_DATAGRAM	MQFB_NONE
MQMSG_CLASS_ACK_REACH_QUEUE	MQMT_REPORT	MQFB_COA
MQMSG_CLASS_ACK_RECEIVE		MQFB_COD
MQMSG_CLASS_NACK_RECEIVE_TIMEOUT		MQFB_EXPIRATION
MQMSG_CLASS_NACK_REACH_QUEUE_ TIMEOUT		MQFB_EXPIRATION
MQMSG_CLASS_NACK_Q_EXCEED_QUOTA		MQRC_Q_FULL
MQMSG_CLASS_NACK_ACCESS_DENIED		MQRC_NOT_AUTHORIZED
MQMSG_CLASS_NACK_ERROR		MQFB_APPL_TYPE_ERROR
Any other value		MQFB_NONE

Note that for MQMSG_CLASS_NORMAL is converted to MQMD_REQUEST if the message includes a response queue (PROPID_M_RESP_QUEUE) or if PROPID_M_RESP_QUEUE is missing, NULL, or an empty string.

Message Expiration (PROPID_M_TIME...)

MSMQ provides two message expiration properties, PROPID_M_TIME_TO_REACH_QUEUE and PROPID_M_TIME_TO_BE_RECEIVED, both in units of seconds. MQSeries provides a single expiration field, MQMD.Expiry, whose units are tenths of a second.

MSMQ-MQSeries Bridge converts the values as follows:

Value of MSMQ properties PROPID_M_TIME_TO_REACH_QUEUE and PROPID_M_TIME_TO_BE_RECEIVED	**Converted to MQSeries value of MQMD.Expiry**
Both values are INFINITEa	MQEI_UNLIMITED
One or both values are not INFINITE	10 times the smaller of the two MSMQ values

Note that MSMQ typically interprets INFINITE as 90 days. In practice, the MSMQ-MQSeries Bridge does not apply the INFINITE conversion because MSMQ decrements the values slightly during transmission. If you need an MQMD.Expiry value of exactly MQEI_UNLIMITED, you should send this value in the message extension.

Message Acknowledgment (PROPID_M_ACKNOWLEDGE)

MSMQ-MQSeries Bridge supports the MSMQ and MQSeries acknowledgment mechanisms. You can send an MSMQ message to MQSeries and receive an automatic acknowledgment from the MQSeries Queue Manager.

To do this, set the MSMQ acknowledgment property (PROPID_M_ACKNOWLEDGE) to a value that requests an acknowledgment. Also specify the administration or response queue name (PROPID_M_ADMIN_QUEUE or PROPID_M_RESP_QUEUE), to which the acknowledgment is sent (see the section on Queue Format Names).

MSMQ-MQSeries Bridge converts the acknowledgment property to the MQSeries MQMD.Report field, as listed in the following table. When MQSeries receives the message, it returns the appropriate acknowledgment via MSMQ-MQSeries Bridge.

Value of MSMQ Property PROPID_M_ACKNOWLEDGE	**Converted to MQSeries value of MQMD.Report**
MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE	MQRO_EXCEPTION | MQRO_COAb
MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE	MQRO_EXCEPTION | MQRO_EXPIRATION | MQRO_CODb
MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE	MQRO_EXCEPTION
MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE	MQRO_EXCEPTION | MQRO_EXPIRATIONb
MQMSG_ACKNOWLEDGMENT_NONE	MQRO_NONE

If both PROPID_M_ACKNOWLEDGE and PROPID_M_JOURNAL are included in an MSMQ message, the value of the MQSeries MQMD.Report field is computed by a bitwise or of the values converted from the two MSMQ properties (see Other MSMQ Properties).

If the MSMQ_PROPID_M_ACKNOWLEDGE property has a value of MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE, the value of MQSeries MQMD.Report field is computed by a bitwise or of MQRO_EXCEPTION and MQRO_COA.

If the MSMQ_PROPID_M_ACKNOWLEDGE property has a value of MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE, the value of MQSeries MQMD.Report field is computed by a bitwise or of MQRO_EXCEPTION, MQRO_EXPIRATION,

and MQRO_COD.

If the MSMQ PROPID_M_ACKNOWLEDGE property has a value of MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE, the value of MQSeries MQMD.Report field is computed by a bitwise or of MQRO_EXCEPTION and MQRO_EXPIRATION.

Other MSMQ Properties with Equivalent Properties

The following MSMQ properties have MQSeries equivalents. The MSMQ-MQSeries Bridge converts the values of each property as listed in the table.

To save space in the table, the prefixes PROPID_M_ and MQMD are omitted from the MSMQ property names and the MQSeries field names, respectively. For example, the first row of data means that PROPID_M_BODY_TYPE is converted to MQMD.Format.

MSMQ property		**Converted to MQSeries**	
PROPID_M_	Value	MQMD.	Value

BODY_TYPE	VT_BSTR Any other value	Format	MQFMT_STRING MQFMT_NONE
CORRELATIONID	Value (20 bytes)	CorrelId	"FQ2Q" + value
DELIVERY	MQMSG_DELIVERY_ RECOVERABLE EXPRESS	Persistence	MQPER_ PERSISTENT NOT_PERSISTENT
JOURNAL	MQMSG_ DEADLETTER MQMSG_JOURNAL MQMSG_JOURNAL_NONE	Reporta	MQRO_ DEAD_LETTER_Q (Ignored) DISCARD_MSG
LABEL LABEL_LEN	Value	ApplIdentityData	Value (MQCHAR32)
MSGID	Value (20 bytes)	MsgId	"FQ2Q" + value
PRIORITY	0 1 2 3 4 5 6 7	Priority	1 3 4 5 6 7 8 9
SENTTIME	Seconds since Jan. 1, 1970	PutDate PutTime	YYYMMDD format HHMMSSTH format

If both PROPID_M_ACKNOWLEDGE and PROPID_M_JOURNAL are included in an MSMQ message, the MQSeries MQMD.Report field is computed by a bitwise or of the values converted from the two MSMQ properties (see Message Acknowledgement).

Unconverted Properties

The following MSMQ properties have no equivalents in MQSeries. The MSMQ-MQSeries Bridge ignores these properties and does not transmit them to MQSeries.

- PROPID_M_APPSPECIFIC
- PROPID_M_ARRIVEDTIME
- PROPID_M_AUTH_LEVEL
- PROPID_M_AUTHENTICATED
- PROPID_M_CONNECTOR_TYPE
- PROPID_M_DEST_SYMM_KEY
- PROPID_M_DEST_SYMM_KEY_LEN
- PROPID_M_ENCRYPTION_ALG
- PROPID_M_HASH_ALG
- PROPID_M_PRIV_LEVEL
- PROPID_M_PROV_NAME
- PROPID_M_PROV_NAME_LEN
- PROPID_M_PROV_TYPE
- PROPID_M_SECURITY_CONTEXT
- PROPID_M_SENDER_CERT
- PROPID_M_SENDER_CERT_LEN
- PROPID_M_SENDERID
- PROPID_M_SENDERID_LEN
- PROPID_M_SENDERID_TYPE
- PROPID_M_SIGNATURE
- PROPID_M_SIGNATURE_LEN

- PROPID_M_SRC_MACHINE_ID
- PROPID_M_TRACE
- PROPID_M_VERSION

Transaction Properties

The following MSMQ properties are not converted to MQSeries fields or values.

- PROPID_M_XACT_STATUS_QUEUE
- PROPID_M_XACT_STATUS_QUEUE_LEN

See the section on Transaction Support Using MSMQ-MQSeries Bridge for more information on transactions.

Building an MQSeries Message

This section explains how the MSMQ-MQSeries Bridge builds a complete MQSeries message, including all needed fields whether or not they have MSMQ equivalents. The conversion to MQSeries fields from MSMQ properties is listed from the MQSeries perspective.

For information on these same conversion rules from the MSMQ perspective, see the previous section on Converting MSMQ Properties.

Message Buffer

The **MQPUT** or **MQGET** message buffer of MQSeries is equivalent to the message body property of MSMQ. The length of the **MQPUT** buffer is the MSMQ message body size.

The following table lists the MQSeries fields and the MSMQ properties from which they are converted.

MQSeries Fields	Converted from MSMQ Property
Message buffer	PROPID_M_BODY
Message buffer length	PROPID_M_BODY_SIZE

Object Descriptor (MQOD)

The MSMQ-MQSeries Bridge retrieves the MQSeries object descriptor fields from the MSMQ format name of the destination queue.

The following table lists the MQSeries fields and the MSMQ properties from which they are converted.

MQSeries Fields	Converted from MSMQ Property
MQOD.ObjectName and MQOD.ObjectQMgrName	PROPID_M_DEST_QUEUE and PROPID_M_DEST_QUEUE_LEN

Message Descriptor (MQMD)

Some fields of the MQSeries message descriptor have no equivalent in MSMQ. The MSMQ-MQSeries Bridge assigns default values to these fields. Alternatively, you can pass explicit values of the MQMD fields using the MSMQ message extension property (PROPID_M_EXTENSION).

The following table lists the MQSeries fields and the default value assigned by MSMQ-MQSeries Bridge.

MQSeries MQMD Fields	Default Value Assigned by MSMQ-MQSeries Bridge
MQMD.AccountingToken	MQACT_NONE
MQMD.CodeCharSetId	MQCCSI_Q_MGR
MQMD.Encoding	MQENC_NATIVE
MQMD.PutApplName	NULL
MQMD.PutApplType	NULL
MQMD.StrucId	MQMD_STRUC_ID
MQMD.Version	MQMD_VERSION_1

Some fields of the MQSeries message descriptor have no equivalent in MSMQ and the MSMQ-MQSeries Bridge does not assign default values to these fields. You can assign explicit values for these MQMD fields using the MSMQ message extension property (PROPID_M_EXTENSION).

The following table lists the MQSeries fields for which no default value is assigned by MSMQ-MQSeries Bridge.

MQSeries MQMD Fields	Default Value Assigned by MSMQ-MQSeries Bridge
MQMD.UserIdentifier	No default value is assigned

Many of the MQMD fields are equivalent to one or more MSMQ properties. The conversion rules are listed in the following table.

You can override the conversion rules by passing explicit MQMD values in the MSMQ message extension property.

To save space in the table, the prefix MQMD. is omitted from the MQSeries field names. For example, the first row of data means that MQMD.ApplIdentityData is built from the MSMQ properties PROPID_M_LABEL and PROPID_M_LABEL_LEN.

MQSeries MQMD Field	MQSeries MQMD Field Value	Converted From MSMQ Property
ApplIdentity	Value (MQCHAR32)	The value of PROPID_M_LABEL and PROPID_M_LABEL_LEN
CorrelId	"FQ2Q" + value	The value of PROPID_M_CORRELATIONID (20 bytes).
Expiry	MQEI_UNLIMITED	Both PROPID_M_TIME_TO_BE_RECEIVED and PROPID_M_TIME_TO_REACH_QUEUE have value of INFINITE
Expiry	10 times the smaller of the two MSMQ values	PROPID_M_TIME_TO_BE_RECEIVED or PROPID_M_TIME_TO_REACH_QUEUE have a value that is not INFINITE
Feedback	MQFB_EXPIRATION	PROPID_M_CLASS has a value of MQMSG_CLASS_NACK_RECEIVE_TIMEOUT
Feedback	MQFB_APPL_TYPE_ERROR	PROPID_M_CLASS has a value of MQMSG_CLASS_NACK_REACH_QUEUE_TIMEOUT
Feedback	MQRC_Q_FULL	PROPID_M_CLASS has a value of MQMSG_CLASS_NACK_Q_EXCEED_QUOTA
Feedback	MQRC_NOT_AUTHORIZED	PROPID_M_CLASS has a value of MQMSG_CLASS_NACK_ACCESS_DENIED
Feedback	MQFB_COA	PROPID_M_CLASS has a value of MQMSG_CLASS_ACK_REACH_QUEUE
Feedback	MQFB_COD	PROPID_M_CLASS has a value of MQMSG_CLASS_ACK_RECEIVE
Feedback	MQFB_NONE	PROPID_M_CLASS has any other value
Format	MQFMT_STRING	PROPID_M_BODY_TYPE has a value of VT_BSTR
Format	MQFMT_NONE	PROPID_M_BODY_TYPE has any other value
MsgId	"FQ2Q" + value	The value of PROPID_M_MSGID (20 bytes)
MsgType	MQMT_DATAGRAM	PROPID_M_CLASS has a value of MQMSG_CLASS_NORMAL
MsgType	MQMT_REQUESTREPORT	PROPID_M_CLASS has any other value
Persistence	MQPER_PERSISTENT	PROPID_M_DELIVERY has a value of MQMSG_DELIVERY_RECOVERABLE
Persistence	MQPER_NOT_PERSISTENT	PROPID_M_DELIVERY has a value of MQMSG_DELIVERY_EXPRESS
Priority	1	PROPID_M_PRIORITY has a value of 0
Priority	3	PROPID_M_PRIORITY has a value of 1
Priority	4	PROPID_M_PRIORITY has a value of 2
Priority	5	PROPID_M_PRIORITY has a value of 3
Priority	6	PROPID_M_PRIORITY has a value of 4
Priority	7	PROPID_M_PRIORITY has a value of 5
Priority	8	PROPID_M_PRIORITY has a value of 6
Priority	9	PROPID_M_PRIORITY has a value of 7
PutDate	YYYMMDD format	The date from the MSMQ PROPID_M_SENTTIME property which has a value of seconds since Jan. 1, 1970
PutTime	HHMMSSTH format	The time from the MSMQ PROPID_M_SENTTIME property which has a value of seconds since Jan. 1, 1970
ReplyToQ	Retrieved from MSMQ format name	The MSMQ format name from PROPID_M_RESP_QUEUE if this property is included and is not NULL or an empty string.
ReplyToQ	Retrieved from MSMQ format name	The MSMQ format name from PROPID_M_ADMIN_QUEUE if PROPID_M_RESP_QUEUE is not included or is NULL or an empty string.
ReplyToQMgr	Retrieve from MSMQ format name	The MSMQ format name from PROPID_M_RESP_QUEUE if this property is included and is not NULL or an empty string.
ReplyToQMgr	Retrieve from MSMQ format name	The MSMQ format name from PROPID_M_ADMIN_QUEUE if PROPID_M_RESP_QUEUE is not included or is NULL or an empty string.
Report	MQRO_EXCEPTION|COA	PROPID_M_ACKNOWLEDGE has a value of MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
Report	MQRO_EXCEPTION|EXPIRATION|COD	PROPID_M_ACKNOWLEDGE has a value of MQMSG_ACKNOWLEDGMENT_FULL_RECEIVE
Report	MQRO_EXCEPTION	PROPID_M_ACKNOWLEDGE has a value of MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
Report	MQRO_EXCEPTION| EXPIRATION	PROPID_M_ACKNOWLEDGE has a value of MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
Report	MQRO_NONE	PROPID_M_ACKNOWLEDGE has a value of MQMSG_ACKNOWLEDGMENT_NONE
Report	MQRO_DEAD_LETTER_Q	PROPID_M_JOURNAL has a value of MQMSG_DEADLETTER

Report	The MSMQ value is ignored	PROPID_M_JOURNAL has a value of MQMSG_JOURNAL
Report	MQRO_DISCARD_MSG	PROPID_M_JOURNAL has a value of MQMSG_JOURNAL_NONE

In practice, the MSMQ-MQSeries Bridge does not assign MQEI_UNLIMITED as value for MQMD.Expiry because MSMQ interprets INFINITE values typically as 90 days and decrements them slightly during transmission. To assign an MQMD.Expiry value of exactly MQEI_UNLIMITED, send this value in the message extension.

The MSMQ-MQSeries Bridge assigns MQMT_DATAGRAM as value for MQMD.MsgType if the MSMQ PROPID_M_RESP_QUEUE is missing, NULL, or an empty string. Otherwise a value of MQMT_REQUEST is assigned to MQMD.MsgType.

If both PROPID_M_ACKNOWLEDGE and PROPID_M_JOURNAL are included in an MSMQ message, MQMD.Report is computed by a bitwise or of the values converted from the two MSMQ properties.

Converting Messages Sent from MQSeries to MSMQ

When you send a message from IBM MQSeries to MSMQ, the MSMQ-MQSeries Bridge converts the message from an MQSeries data structure to an MSMQ message property. In order to do this, MSMQ-MQSeries Bridge maps the various data fields of the MQSeries message as nearly as possible to equivalent MSMQ message properties.

This section describes the conversion rules by which this is done. The information in this section applies to messages that you send from MQSeries to MSMQ. For messages sent from MSMQ to MQSeries, see the preceding section on Converting Messages Sent from MSMQ to MQSeries.

The section is divided into two main subsections, which provide essentially the same information but from complementary points of view. The first section, Converting MQSeries Fields, describes the conversion rules from the sender's point of view. This section explains how the MSMQ-MQSeries Bridge converts each MQSeries field that you include in a message. The second section, Building an MSMQ Message, explains the rules from the receiver's point of view. Read this section to learn how MSMQ-MQSeries Bridge builds a complete MSMQ message containing all the needed properties, whether or not they have exact MQSeries equivalents.

Besides the conversions described here, the MSMQ-MQSeries Bridge transmits the original MQSeries message descriptor fields in the MSMQ message extension property (PROPID_M_EXTENSION). For information on this subject, see the section on Message Extension.

This section contains:

Converting MQSeries Fields

This section explains how the MSMQ-MQSeries Bridge converts the fields of an MQSeries message to MSMQ. For information on MSMQ properties that have no MQSeries equivalents, see the section below on Building an MSMQ Message.

Message Buffer

The **MQGET** or **MQPUT** buffer of MQSeries is equivalent to the message body property of MSMQ.

The following table lists the MQSeries fields and the MSMQ properties that they are converted to.

MQSeries Fields	Converted to MSMQ Property
Message buffer	PROPID_M_BODY
Message buffer length	PROPID_M_BODY_SIZE

Object Descriptor (MQOD)

The MSMQ-MQSeries Bridge converts the MQSeries remote queue manager and queue names to the MSMQ destination queue name. The MQSeries fields are interpreted as an MSMQ format name or path name, as described below.

The following table lists the MQSeries fields and the MSMQ properties that they are converted to.

MQSeries Fields	Converted to MSMQ Property
MQOD.ObjectName and MQOD.ObjectQMgrName	PROPID_M_DEST_QUEUE and PROPID_M_DEST_QUEUE_LEN

Character Substitutions in Object Descriptor Conversion

Certain characters are supported in MSMQ format names but not in MQSeries names. When you assign the MQSeries MQOD.ObjectName, the MSMQ-MQSeries Bridge performs character substitutions to the MSMQ format names. The table below lists special characters in MQSeries names and what these characters are converted to in MSMQ format names by the MSMQ-MQSeries Bridge.

Characters in MQSeries MQOD Field Names	Characters Substituted in MSMQ Format Names
/ (forward slash))	: (colon) or \ (backslash)
_ (underscore)	- (hyphen) or = (equal)
P_ (at start of MQOD.ObjectName)	PRIVATE=

The MSMQ-MQSeries Bridge converts the characters back when it transmits the MQSeries message to MSMQ. The characters are interpreted according to context to generate a legal MSMQ name. For example, MSMQ-MQSeries Bridge converts the format name

```
DIRECT_OS/MACHINE2/QUEUE4
```

to

```
DIRECT=OS:MACHINE2\QUEUE4.
```

Note that this character substitution is only for format names, not for machine or path names. If you want to address MSMQ path names, do not include hyphens or other characters not supported by MQSeries. For example, the path name

```
MACHINE2\MY-QUEUE
```

is legal in MSMQ, but you cannot specify the hyphen character in MQSeries.

Format Name Method of Object Descriptor Conversion

Subject to the following conditions, the MSMQ-MQSeries Bridge interprets the MQSeries MQOD.ObjectName field as the format

name of the MSMQ destination queue:

The MQOD.ObjectQMgrName is an MQSeries alias for an MSMQ machine.

The MQOD.ObjectName begins with PUBLIC_, P_, or DIRECT_OS/.

If these conditions apply, the MSMQ-MQSeries Bridge converts the MQSeries MQOD.ObjectName to the MSMQ format name of the destination queue using one of the following methods based on the value of MQOD.ObjectName:

```
PUBLIC=<GUID>
PRIVATE=<machine GUID>\<file number>
DIRECT=OS:<path name>.
```

If you know the GUID of the destination queue, the PUBLIC=<GUID> syntax gives better performance than the DIRECT=OS:<path name> syntax. Also note that non-transacted private queues are supported only on the MSMQ-MQSeries Bridge computer.

Path Name Method of Object Descriptor Conversion

If the conditions for the format name method do not hold, the MSMQ-MQSeries Bridge interprets MQOD.ObjectQMgrName\MQOD.ObjectName as an MSMQ path name. The remote queue manager name must be an alias for an MSMQ machine that you have previously defined in MQSeries.

For example, if MQOD.ObjectQMgrName is MACHINE2 and MQOD.ObjectName is QUEUE4, the MSMQ-MQSeries Bridge sends the message to an MSMQ queue having the path name MACHINE2\QUEUE4.

The MSMQ-MQSeries Bridge determines the MSMQ format name corresponding to this path name in order to forward the message.

Note that the MQSeries name fields are limited to 48 characters each. If the path name is longer than this, use the format name method instead.

Queue Alias Method of Object Descriptor Conversion

Optionally, you can address an MSMQ queue using an MQSeries queue alias. If you use this method, set MQOD.ObjectName to the queue alias. Leave MQOD.ObjectQMgrName blank, or set it to the MQSeries Queue Manager where the transmission queue is located.

Examples of Object Descriptor Conversion

In the following examples, you want to send a message to an MSMQ destination having the following identifiers:

```
Machine name = MACHINE2
Queue name = QUEUE4
GUID = A56F41B4-9869-11D0-AF8F-0000E8D1C3A7
```

MSMQ-MQSeries Bridge is installed on a machine called BRIDGEMQ1. You have defined the aliases BRIDGEMQ1 and BRIDGEMQ1% for this machine in MQSeries. In the MSMQ-MQSeries Bridge Manager, you have configured BRIDGEMQ1 for MQS->MSMQ message pipe and BRIDGEMQ1% for MQS->MSMQ transactional message pipe.

You may address a message to this queue in any of the following ways:

Format Name method

Using the format name method and MQS->MSMQ message pipe, the MQSeries name would become:

```
MQOD.ObjectQMgrName = "BRIDGEMQ1"
MQOD.ObjectName = "PUBLIC_A56F41B4_9869_11D0_AF8F_0000E8D1C3A7"
```

-or-

```
MQOD.ObjectQMgrName = "BRIDGEMQ1"
```

```
MQOD.ObjectName = "DIRECT_OS/MACHINE2/QUEUE4"
```

Using the format name method and MQS->MSMQ transactional message pipe, the MQSeries name would become

```
MQOD.ObjectQMgrName = "BRIDGEMQ1%"
MQOD.ObjectName = "PUBLIC_A56F41B4_9869_11D0_AF8F_0000E8D1C3A7"
```

-or-

```
MQOD.ObjectQMgrName = "BRIDGEMQ1%"
MQOD.ObjectName = "DIRECT_OS/MACHINE2/QUEUE4"
```

Optionally, you can define the MQSeries aliases MACHINE2 and MACHINE2% for the MSMQ destination machine. You now have two additional ways to address the queue:

Using the path name method and MQS->MSMQ message pipe, the MQSeries name would become

```
MQOD.ObjectQMgrName = "MACHINE2"
MQOD.ObjectName = "QUEUE4"
```

Using the path name method and MQS->MSMQ transactional message pipe, the MQSeries name would become

```
MQOD.ObjectQMgrName = "MACHINE2%";
MQOD.ObjectName = "QUEUE4"
```

In yet another option, you can define the MQSeries aliases QUEUE4 and QUEUE4% for the MSMQ destination queues. If you do this, you can address the queue using the following syntax:

Using the queue alias method and normal service, the MQSeries name would become

```
MQOD.ObjectQMgrName = ""
MQOD.ObjectName = "QUEUE4"
```

Using the queue alias method and high service, the MQSeries name would become

```
MQOD.ObjectQMgrName = ""
MQOD.ObjectName = "QUEUE4%"
```

Message Descriptor (MQMD)

The MSMQ-MQSeries Bridge converts most of the MQSeries Field values to MSMQ property values.

MQMD.Report Field

The MSMQ-MQSeries Bridge supports the MQSeries and MSMQ acknowledgment mechanisms. You can send an MSMQ message to MQSeries and receive an automatic acknowledgment from the MSMQ Queue Manager.

To do this, set the MQMD.Report field of the MQSeries message to a value that requests an acknowledgment. Also set the MQMD.ReplyToQ and MQMD.ReplyToQMgr fields, which specify where the acknowledgment is sent.

The MSMQ-MQSeries Bridge converts MQMD.Report to the MSMQ acknowledgment property. When MSMQ receives the message, it returns the appropriate acknowledgment via the MSMQ-MQSeries Bridge.

The MSMQ-MQSeries Bridge also supports the MQSeries and MSMQ dead letter mechanism. For this purpose, MSMQ-MQSeries Bridge converts the MQMD.Report values to the MSMQ journaling property.

The conversions are listed in the following table.

Value of MQSeries Field MQMD.Report	**Converted to MSMQ Property an Value**

MQRO_NONE	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NONE
MQRO_EXCEPTION	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
MQRO_EXCEPTION_WITH_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
MQRO_EXCEPTION_WITH_FULL_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_REACH_QUEUE
MQRO_EXPIRATION	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
MQRO_EXPIRATION_WITH_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
MQRO_EXPIRATION_WITH_FULL_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_NACK_RECEIVE
MQRO_COA	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
MQRO_COA_WITH_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
MQRO_COA_WITH_FULL_DATA (Note 1)	The value of PROPID_M_ACKNOWLEDGE is set to MQMSG_ACKNOWLEDGMENT_FULL_REACH_QUEUE
MQRO_DEAD_LETTER_Q	The value of PROPID_M_JOURNAL is set to MQMSG_DEADLETTER
MQRO_DISCARD_MSG	The value of PROPID_M_JOURNAL is set to MQMSG_JOURNAL_NONE
MQRO_NEW_MSG_ID	(Not converted)
MQRO_PASS_MSG_ID	(Not converted)
MQRO_COPY_MSG_ID_TO_CORREL_ID	(Not converted)
MQRO_PASS_CORREL_ID	(Not converted)

Note For these report values, the MSMQ-MQSeries Bridge duplicates part or all of the message buffer in the MSMQ message extension (PROPID_M_EXTENSION). The increased message size may degrade performance.

MQMD.MsgType and MQMD.Feedback Fields

The values of the MQSeries MQMD.MsgType and MQMD.Feedback fields are converted to the MSMQ message class property.

The conversions are listed in the following table based on the value of MQMD.MsgType when MQMD.Feedback is MQFB_NONE.

Value of MQSeries MQMD.MsgType	**Converted to Value of MSMQ PROP_M_CLASS**
MQMT_SYSTEM_FIRST	MQMSG_CLASS_NORMAL
MQMT_SYSTEM_LAST	MQMSG_CLASS_NORMAL
MQMT_DATAGRAM	MQMSG_CLASS_NORMAL
MQMT_REQUEST	MQMSG_CLASS_NORMAL
MQMT_REPLY	MQMSG_CLASS_NORMAL
MQMT_APPL_FIRST	MQMSG_CLASS_NORMAL
MQMT_APPL_LAST	MQMSG_CLASS_NORMAL

The conversions are listed in the following table based on the value of MQMD.Feedback when MQMD.MsgType is MQMT_REPORT.

Value of MQSeries MQMD.Feedback	**Converted to Value of MSMQ PROP_M_CLASS**
MQFB_EXPIRATION	MQMSG_CLASS_NACK_RECEIVE_TIMEOUT
MQFB_COA	MQMSG_CLASS_ACK_REACH_QUEUE
MQFB_COD	MQMSG_CLASS_ACK_RECEIVE
MQFB_APPL_TYPE_ERROR	MQMSG_CLASS_NACK_ERROR
MQFB_DATA_LENGTH_ZERO	MQMSG_CLASS_NACK_ERROR
MQFB_DATA_LENGTH_NEGATIVE	MQMSG_CLASS_NACK_ERROR
MQFB_DATA_LENGTH_TOO_BIG	MQMSG_CLASS_NACK_ERROR
MQFB_BUFFER_OVERFLOW	MQMSG_CLASS_NACK_ERROR
MQFB_LENGTH_OFF_BY_ONE	MQMSG_CLASS_NACK_ERROR

MQFB_NONE	(Not converted)
MQFB_SYSTEM_FIRST	(Not converted)
MQFB_SYSTEM_LAST	(Not converted)
MQFB_APPL_FIRST	(Not converted)
MQFB_APPL_LAST	(Not converted)
MQFB_TM_ERROR	(Not converted)
MQFB_IIH_ERROR	(Not converted)
MQFB_NOT_AUTHORIZED_FOR_IMS	(Not converted)
MQFB_IMS_ERROR	(Not converted)
MQFB_IMS_FIRST	(Not converted)
MQFB_IMS_LAST	(Not converted)
MQFB_QUIT	(Not supported)
MQRC_NOT_AUTHORIZED	MQMSG_CLASS_NACK_ACCESS_DENIED
MQRC_Q_FULL	MQMSG_CLASS_NACK_Q_EXCEED_QUOTA
MQRC_PERSISTENT_NOT_ALLOWED	MQMSG_CLASS_NACK_ERROR
MQRC_MSG_TOO_BIG_FOR_Q_MGR	MQMSG_CLASS_NACK_ERROR
MQRC_MSG_TOO_BIG_FOR_Q	MQMSG_CLASS_NACK_ERROR
MQRC_PUT_INHBITED	(Not converted)

MQMD.ReplyToQ and MQMD.ReplyToQMgr Fields

The MSMQ-MQSeries Bridge converts the MQSeries ReplyToQMgr and ReplyToQ fields to an MSMQ format name. The name is assigned both to the response queue property and to the administration queue property of the new MSMQ message.

The MSMQ-MQSeries Bridge interprets the ReplyToQMgr and the ReplytoQ fields in the same way as the destination queue name:

If MQMD.ReplyToQMgr is the MSMQ-MQSeries Bridge machine name and MQMD.ReplyToQ begins with PUBLIC_, P_, or DIRECT_OS/, MSMQ-MQSeries Bridge interprets MQMD.ReplyToQ as an MSMQ format name.

Otherwise, the MSMQ-MQSeries Bridge interprets MQMD.ReplyToQMgr\MQMD.ReplyToQ as an MSMQ path name and determines the MSMQ format name.

For information on character substitutions within the names and other syntax information, see the section on Object Descriptor.

The following table lists the MQSeries fields and the MSMQ properties that they are converted to.

MQSeries Fields	**Converted to MSMQ Property**
MQOD.ReplyToQMgr and MQOD.ReplyToQ	PROPID_M_RESP_QUEUE
MQOD.ReplyToQMgr and MQOD.ReplyToQ	PROPID_M_ADMIN_QUEUE (same value as PROPID_M_RESP_QUEUE)

Other MQMD Fields

The following table lists the conversions of additional MQMD fields, besides the ones described above, to MSMQ properties.

> **Note** To save space in the table, the prefixes **MQMD.** and **PROPID_M_** are omitted from the MQSeries field names and the MSMQ property names, respectively. For example, the first row of data means that **MQMD.ApplIdentityData** is converted to **PROPID_M_LABEL** and **PROPID_M_LABEL_LEN**.

MQSeries field		**Converted to MSMQ**	
MQMD.	Value	PROPID_M_	Value
ApplIdentityData	Value (MQCHAR32)	LABEL	Value
CorrelId	Value (MQBYTE24)	CORRELATIONID	Last 20 bytes of value
Expiry	MQEI_UNLIMITED Value > 0 (tenths of seconds)	TIME_TO_BE_RECEIVED	INFINITE Value/10 (seconds)
Format	MQFMT_STRING Any other value	BODY_TYPE	VT_BSTR Not converted

Persistence	MQPER_ PERSISTENT NOT_PERSISTENT	DELIVERY	MQMSG_DELIVERY_ RECOVERABLE EXPRESS
Priority	0, 1 2, 3 4 5 6 7 8 9	PRIORITY	0 1 2 3 4 5 6 7
UserIdentifier	Value	SENDERID	SID value (if the user is registered in Windows NT) Not converted (if the user is not registered)

Unconverted MQSeries MQMD Fields

The following MQSeries fields have no equivalents in MSMQ. The MSMQ-MQSeries Bridge ignores these fields and does not transmit them to MSMQ.

Like all MQMD fields, MSMQ-MQSeries Bridge stores the fields in the MSMQ message extension property.

- MQMD.AccountingToken
- MQMD.ApplOriginData
- MQMD.BackoutCount
- MQMD.Encoding
- MQMD.MsgId
- MQMD.PutApplName
- MQMD.PutApplType
- MQMD.PutDate
- MQMD.PutTime
- MQMD.StrucId

When the value of MQMD.CodeCharSetID is MQCCSI_Q_QMG, the MSMQ-MQSeries Bridge ignores this field and does not transmit it to MSMQ. When the value of MQMD.CodeCharSetID is MQCCSI_Q_QMG, this value is not supported by the MSMQ-MQSeries Bridge.

When the value of MQMD.Version is "MQPMO_VERSION_1", the MSMQ-MQSeries Bridge ignores this field and does not transmit it to MSMQ.

Building an MSMQ Message

This section explains how the MSMQ-MQSeries Bridge builds a complete MSMQ message, including all needed properties, whether or not they have MQSeries equivalents.

Message Body (**PROPID_M_BODY**)

The MSMQ message body is equivalent to the **MQPUT** or **MQGET** message buffer of MQSeries. The length of the **MQPUT** buffer is the MSMQ message body size.

The following table lists the MSMQ properties and the MQSeries fields from which they are converted.

MSMQ Property	Converted from MQSeries Field
PROPID_M_BODY	Message buffer
PROPID_M_BODY_SIZE	Message buffer length

Queue Format Names (**PROPID_M_..._QUEUE**)

MSMQ-MQSeries Bridge retrieves the MSMQ format names of the destination, response, and administration queues from the MQSeries queue and queue manager names.

The following table lists the MSMQ properties and the MQSeries fields from which they are converted.

MSMQ Properties	Converted from MQSeries Fields
PROPID_M_DEST_QUEUE and PROPID_M_DEST_QUEUE_LEN	MQOD.ObjectName and MQOD.ObjectQMgrName
PROPID_M_RESP_QUEUE and PROPID_M_RESP_QUEUE_LEN	MQMD.ReplyToQ and MQMD.ReplyToQMgr
PROPID_M_ADMIN_QUEUE and PROPID_M_ADMIN_QUEUE_LEN	MQMD.ReplyToQ and MQMD.ReplyToQMgr

Message Class (PROPID_M_CLASS)

The MSMQ-MQSeries Bridge assigns the MSMQ message class based on the MQSeries MQMD.MsgType and MQMD.Feedback values. For detailed information, see the section MQMD.MsgType and MQMD.Feedback Fields.

The following table lists the MSMQ properties and the MQSeries fields from which they are converted.

MSMQ Properties	Converted from MQSeries Fields
PROPID_M_CLASS	MQMD.MsgType and MQMD.Feedback

Message Expiration (PROPID_M_TIME...)

The MSMQ-MQSeries Bridge converts the MQSeries MQMD.Expiry value (in tenths of seconds) to the MSMQ PROPID_M_TIME_TO_BE_RECEIVED (in seconds). If MQMD.Expiry is set to MQEI_UNLIMITED, then the value of MSMQ PROPID_M_TIME_TO_BE_RECEIVED is set to INFINITE. For other values of MQMD.Expiry greater than zero, the value of MSMQ PROPID_M_TIME_TO_BE_RECEIVED is converted by dividing MQMD.Expiry by 10. The MSMQ-MQSeries Bridge does not set PROPID_M_TIME_TO_REACH_QUEUE.

VALUE of MSMQ Properties	Converted from MQSeries Fields
PROPID_M_TIME_TO_BE_RECEIVED	MQMD.Expiry

Message Acknowledgment and Journaling (PROPID_M_ACKNOWLEDGE, PROPID_M_JOURNAL)

The MSMQ message acknowledgment and journaling are converted from values of MQMD.Report. For details, see the section above on MQMD.Report.

MSMQ property	Converted from MQSeries
PROPID_M_ACKNOWLEDGE PROPID_M_JOURNAL	MQMD.Report

Other MSMQ Properties (PROPID_M_...)

Fields without MQSeries equivalents

Some MSMQ properties have no equivalent in MQSeries. The MSMQ-MQSeries Bridge assigns the following values to these

properties.

MSMQ property	Value assigned by MSMQ-MQSeries Bridge
PROPID_M_APPSPECIFIC	None
PROPID_M_AUTH_LEVEL	Default
PROPID_M_CONNECTOR_TYPE	None
PROPID_M_DEST_QUEUE	Default
PROPID_M_DEST_QUEUE_LEN	Default
PROPID_M_DEST_SYMM_KEY	None
PROPID_M_ENCRYPTION_ALG	None
PROPID_M_HASH_ALG	None
PROPID_M_MSGID	Assigned by MSMQ
PROPID_M_PRIV_LEVEL	Default
PROPID_M_PROV_NAME	None
PROPID_M_PROV_TYPE	None
PROPID_M_SECURITY_CONTEXT	Default
PROPID_M_SENDER_CERT	Default
PROPID_M_SENTTIME	Time when MSMQ-MQSeries Bridge transmits the message to MSMQ
PROPID_M_SIGNATURE	None
PROPID_M_SRC_MACHINE_ID	GUID of the MSMQ-MQSeries Bridge machine
PROPID_M_TRACE	Default
PROPID_M_VERSION	0x0010

Equivalent MSMQ Properties

The following MSMQ properties have MQSeries equivalents. The MSMQ-MQSeries Bridge converts the values of each property as listed in the table.

> To save space in the table, the prefixes PROPID_M_ and MQMD. are omitted from the MSMQ property names and the MQSeries field names, respectively. For example, the first row of data means that PROPID_M_BODY_TYPE is converted from MQMD.Format.

MSMQ property		Converted from MQSeries	
PROPID_M_	**Value**	MQMD.	**Value**
BODY_TYPE	VT_BSTR Not converted (default)	Format	MQFMT_STRING Any other value
CORRELATIONID	Last 20 bytes of value	MsgId	Value (MQBYTE24)
DELIVERY	MQMSG_DELIVERY_ RECOVERABLE EXPRESS	Persistence	MQPER_ PERSISTENT NOT_PERSISTENT
LABEL LABEL_LEN	Value	ApplIdentityData	Value (MQCHAR32)
PRIORITY	0 1 2 3 4 5 6 7	Priority	0, 1 2, 3 4 5 6 7 8 9
SENDERID SENDERID_TYPE	SID value MQMSG_SENDERID_TYPE_SID Not converted MQMSG_SENDERID_TYPE_NONE	UserIdentifier	Value, if the user is registered in Windows NT If the user is not registered in Windows NT

The MSMQ-MQSeries Bridge Extensions Mechanism

Besides the automatic conversion of MSMQ (Microsoft Message Queue) and IBM MQSeries messages, the Microsoft® MSMQ-MQSeries Bridge provides a mechanism for sending and receiving explicit MQSeries field values. The Microsoft® MSMQ-MQSeries Bridge Extension Property API supports the message extension property (PROPID_M_EXTENSION) of Microsoft Message Queue Server (MSMQ). The message extension property provides a way for applications to attach any type of data—in essence, custom properties—to an MSMQ message.

Similar to the MSMQ message body property (PROPID_M_BODY), the message extension can have any length. However, the message extension has a defined structure that lets an application label its data with a GUID (Globally Unique Identifier) code. Applications can attach multiple extension fields, each labeled with its own GUID and all included in a single message extension property. This is done using the MSMQ message extension property (PROPID_M_EXTENSION).

This section contains:

Data Structure of a Message Extension

A message extension is an MSMQ message property of arbitrary length. The property symbol of a message extension is PROPID_M_EXTENSION and its type indicator is VT_UI1|VT_VECTOR. A message extension is a sequential buffer containing any number of MSMQ extension fields.

Each extension field comprises three subfields:

Subfield	Length of Subfield (bytes)	Description
GUID	16	A GUID identifier, typically of the application that created the extension field.
Length	4	The Length of the Data subfield in bytes.
Data	Value of the length subfield	Any data that is part of the message extension.

For use with the MSMQ-MQSeries Bridge, the GUID identifier is set to the MSMQ-MQSeries Bridge GUID value.

The MSMQ message extension length property, PROPID_M_EXTENSION_LEN, is of type indicator VT_UI4 and represents the overall size in bytes of all message extensions attached to a message. MSMQ sets the message extension length automatically when you send a message. When you receive or peek at a message, you can look into the message extension length to detect whether the message contains any message extension fields and to determine the necessary receive buffer size.

The MSMQ-MQSeries Bridge Extension API functions work with an alternative data representation for a message extension, called an EP object. An EP object contains the same fields and subfields as a message extension, but in a format adapted for programming.

How MSMQ-MQSeries Bridge Creates a Message Extension

When the MSMQ-MQSeries Bridge processes a message from MQSeries, it creates a `PROPID_M_EXTENSION` property, which it includes in the message that it transmits to MSMQ.

This section contains:

MQMD Extension Field

Ordinarily, the message extension contains a single extension field with the following structure:

- The MSMQ-MQSeries Bridge GUID code is stored in the GUID subfield of the extension.

The sizeof(MQMD) is stored in the length subfield of the extension.

The MQMD is copied byte-for-byte into the data buffer of the extension.

The MSMQ-MQSeries Bridge GUID is the value of the sg_MSMQExtMQMD constant which is defined in the mqsrext.h include file found in the SDK\Include subdirectory.

The MQMD extension has the following GUID for MQMD version 2.

```
static const GUID sg_MSMQExtMQMDE =
{ 0x18ae68f5, 0x989b, 0x11d3,
    { 0x8d, 0xf9, 0x0, 0x0, 0xf8, 0x1a,  0xea, 0x1f }
};
```

Error Extension Field

If the MSMQ-MQSeries Bridge encounters an MQSeries error when it transmits a message from MSMQ, it records the error in the extension property and places the message on the dead letter queue.

To do this, the MSMQ-MQSeries Bridge adds an extension property to the MSMQ message, if it doesn't already exist. Within the extension property, MSMQ-MQSeries Bridge creates an extension field containing the following data:

The GUID subfield contains a MSMQ-MQSeries Bridge error GUID (different from the GUID used for MQMD).

- The length subfield contains the value 4.

The data subfield contains a reason code, identical to the codes returned by the MQSeries function **MQPUT**.

The MSMQ-MQSeries Bridge error GUID is the value of the sg_MSMQExtReasonCode constant, which is defined in the mqsrext.h include file found in the SDK\Include subdirectory.

If desired, an MSMQ application can read messages from the dead letter queue and interpret the reason codes.

Other Extension Fields

If you send an MQSeries message having certain values of MQMD.Report, the MSMQ-MQSeries Bridge adds a second extension field to the new MSMQ message. This field is for internal use only, not for use in your applications. The MSMQ-MQSeries Bridge distinguishes the Report extension field from the MQMD and error extension fields by labeling them with different GUIDs.

For further information, see Converting Messages Sent from MQSeries to MSMQ.

How MSMQ-MQSeries Bridge Converts a Message Extension

If you send an MSMQ message including a message extension to MQSeries, the MSMQ-MQSeries Bridge converts the extension in the following way:

The MSMQ-MQSeries Bridge looks for an extension field identified by the MSMQ-MQSeries Bridge GUID. If it finds one, it reads the MQMD structure from the extension field.

- MSMQ-MQSeries Bridge ignores any other extension fields that may be present in the message extension.

MSMQ-MQSeries Bridge includes the MQMD structure that it reads from the extension field in the new MQSeries message.

The MQMD structure in the message extension overrides the default MQMD conversions, which are described in the section on Converting Messages Sent from MSMQ to MQSeries.

A few exceptions to the above rules are described in the following sections.

This section contains:

- Sender and User Identifiers
- Version Identifiers

Sender and User Identifiers

MSMQ-MQSeries Bridge reads the value of MQMD.UserIdentifier stored in the message extension and assigns the value to the MQMD.UserIdentifier field in the new MQSeries message.

Version Identifiers

MQSeries uses the following fields for version identification:

- MQMD.StrucId
- MQMD.Version

When the MSMQ-MQSeries Bridge converts a message extension, it confirms that the values of these fields are for a version of MQSeries that the Bridge supports. If they are not, MSMQ-MQSeries Bridge places the message on the dead letter queue and does not transmit it to MQSeries.

For the supported versions of MQSeries, see Platforms Supported By MSMQ-MQSeries Bridge Extensions. For the permitted values of the version identification fields, see Converting Messages Sent from MSMQ to MQSeries.

Using Message Extensions

This section suggests a few ways that you can use the message extension property in your messaging applications.

This section contains:

Sending an MQSeries Message to MSMQ

MSMQ-MQSeries Bridge stores the complete MQMD structure of an MQSeries message in an MSMQ message extension. An MSMQ application can read the extension and retrieve the original MQMD structure.

In this way, an MSMQ application can retrieve the original values of every MQMD field, regardless of the MSMQ-MQSeries Bridge conversions.

Sending an MSMQ Message to MQSeries

When you send an MSMQ message to MQSeries, you can include a message extension. This can be an extension that you originally received from MQSeries, or one that you created yourself. MSMQ-MQSeries Bridge reads the message extension and uses it to set the MQMD fields of the converted message that it sends to MQSeries.

You can use this feature for two purposes:

- To override the default conversions described in the section Converting Messages Sent from MSMQ to MQSeries

To supplement the default conversions by assigning MQMD fields that have no MSMQ equivalent

The following are some examples of fields that have no equivalents or only partial equivalents (different permitted values or length) among the MSMQ message properties:

- MQMD.AccountingToken
- MQMD.ApplOriginData
- MQMD.CorrelId
- MQMD.MsgId
- MQMD.MsgType
- MQMD.PutApplName
- MQMD.PutApplType
- MQMD.ReplyToQ
- MQMD.ReplyToQMgr
- MQMD.Report

Suppose you want to send a message to an MQSeries application including an MQMD.MsgType value of MQMD_REPLY. The default message conversions provide no way to set this particular value. You can send the value by storing an MQMD data structure in a message extension.

Programming a Message Extension

In an MSMQ application, you can program a message extension containing an arbitrary number of extension fields. For use with MSMQ-MQSeries Bridge, build the extension according to the following specifications.

It is recommended that you use MSMQ-MQSeries Bridge Extension Property API to construct the extension with the required syntax. Create a PROPID_M_EXTENSION property containing at least one extension field.

Store the MSMQ-MQSeries Bridge GUID code in the GUID subfield of exactly one extension field. The GUID is the value of the sg_MSMQExtMQMD constant, which is defined in the mqsrext.h include file of the Extension Property API. Store the sizeof(MQMD) in the length subfield. Copy a complete MQMD structure byte-for-byte into the data buffer subfield.

When you send the message to MQSeries, the MSMQ-MQSeries Bridge converts the extension field identified by the MSMQ-MQSeries Bridge GUID. Any other extension fields are ignored.

The MSMQ-MQSeries Bridge Extension Property API

The MSMQ-MQSeries Bridge Extension Property API is recommended for programming and working with message extensions. The API provides a library of functions that help you create and interpret message extensions. The MSMQ-MQSeries Bridge Extension Property API is supplied as part of the Host Integration Server SDK.

Your MSMQ applications can use the API to:

- Read message extensions that you receive from MQSeries
- Create or modify message extensions that you send to MQSeries
- Read or create message extensions for any other MSMQ messaging purpose

The MSMQ-MQSeries Bridge Extension Property API lets you create and work with the MSMQ message extension property easily. This section provides a few guidelines for using the API functions.

The MSMQ-MQSeries Bridge Extension Property API operates directly on the EP representation of a message extension. In particular, the API functions support the following programming approach:

- Creating or deleting an EP object.
- Creating, finding, reading, writing, or deleting extension fields in an EP object.

Converting an EP object to a message extension. (PROPID_M_EXTENSION) that you can send in an MSMQ message.

- Converting a message extension that you received in an MSMQ message to an EP object.

You cannot send an EP object directly in an MSMQ message. You must first convert it to a message extension (PROPID_M_EXTENSION).

To use the message extension API, include the `MSMQext.h` header file in your applications. This header files contains constants and function prototypes for the Extension Property API functions. This header file is located in the SDK\Include directory on the Host Integration Server CD-ROM and is installed when the SDK package is selected.

If you are using the API in conjunction with MSMQ-MQSeries Bridge, also include the mqsrext.h header file located in the SDK\Include directory. This file defines the GUID that labels the extension fields.

The Extension Property API functions are handle-based. The following is a summary of the MSMQ-MQSeries Bridge Extension Property API functions. For complete details, see the individual function descriptions in the MSMQ-MQSeries Bridge Extensions Reference.

Function	Description
EPAdd	Adds a new extension field to an EP object.
EPClose	Frees the extension handle and associated memory of an EP object.
EPDelete	Deletes an extension field from an EP object.
EPDeleteAll	Deletes all extension fields or all extensions fields matching a specific GUID from an EP object.
EPGet	Positions to and optionally retrieves a requested extension field from an EP object, storing the GUID, length, and data subfields in separate variables. **EPGet** can also be used to locate extension fields containing a specified GUID.
EPGetBuffer	Converts an EP object to a message extension and packs the message extension into the supplied buffer.
EPOpen	Creates an EP object and optionally unpacks the supplied message extension buffer into it.
EPUpdate	Writes new data to an existing extension field of an EP object.

Programming Considerations Using MSMQ-MQSeries Bridge Extensions

The limitations of specific API functions results from the fact that MSMQ-MQSeries Bridge transmits messages, not API calls, between queuing systems. The Microsoft MSMQ-MQSeries Bridge transmits messages across the MSMQ-MQSeries interface. The MSMQ-MQSeries Bridge does not transmit API calls across the interface. Thus, MSMQ API calls operate only within the MSMQ environment, and MQSeries API calls operate only within the MQSeries environment. This principle limits the ways you can create and access queues.

All MSMQ API functions operate only within the MSMQ environment, up to and including foreign computers and queues. For example, you can use the following functions:

MQLocateBegin, **MQLocateNext**, and **MQLocateEnd** to search for foreign queues

MQGetMachineProperties, **MQGetPrivateComputerInformation**, **MQGetQueueProperties**, and **MQGetQueueSecurity** functions to retrieve the properties of foreign queues

MQOpenQueue to open a foreign queue

MQSetQueueProperties and **MQSetQueueSecurity** to set the properties of foreign queues

MQCloseQueue to close a foreign queue

When creating a queue, you can call the MSMQ function **MQCreateQueue** to create a foreign queue representing an MQSeries queue, but you cannot create the actual MQSeries queue itself. Similarly, you cannot create an MSMQ Queue by calling the MQSeries function **MQOPEN**.

To communicate across the MSMQ-MQSeries interface, your MSMQ and MQSeries applications should each create their own queues. Alternatively, you can use administration tools such as the MSMQ Manager or the MQSeries command interface to create the queues.

For proper message delivery, you must ensure that the destination queue for each message actually exists.

You can use the MSMQ **MQPathNameToFormatName** function to determine an MSMQ format name for an MQSeries queue. The format name actually refers to the MSMQ foreign queue. The MSMQ-MQSeries Bridge processes the format name that it finds in a message and directs the message to the MQSeries queue.

When opening a queue, a call to the MSMQ function **MQOpenQueue** opens the foreign queue, not the MQSeries queue itself. The MSMQ-MQSeries Bridge opens the MQSeries queue as necessary when it transmits a message. If you are sending messages to more that one MQSeries queue, you must open each one separately using its own MSMQ format name.

In the opposite direction, the MQSeries function **MQOPEN** opens the transmission queue for the MSMQ machine. The MSMQ-MQSeries Bridge opens the MSMQ queue when it transmits a message.

When sending an MSMQ message to a foreign queue with **MQSendMessage**, MSMQ delivers the message to the connector queue in the MSMQ-MQSeries Bridge machine. The MSMQ-MQSeries Bridge converts and transmits the message to MQSeries queue. MQSeries delivers a message sent by **MQPUT** to a transmission queue. The MSMQ-MQSeries Bridge reads the message from the MQSeries transmission queue. After converting the message from MQSeries to MSMQ message properties, MSMQ-MQSeries Bridge transmits the message to the destination MSMQ queue.

When receiving a message, the MSMQ-MQSeries Bridge does not transmit receive requests across the MSMQ-MQSeries interface. An MSMQ application can receive a message only from a native MSMQ queue (the **MQReceiveMessage** function). An MQSeries application can receive only from a native MQSeries queue (the **MQGET** function).

When sending a message from MQSeries to MSMQ, if you want the MQSeries message to have a value for MQMD. ApplIdentityData, you need the set both of the following:

- Set the open option with MQOO_SET_IDENTITY_CONTEXT
- Set the put option with MQPMO_SET_IDENTITY_CONTEXT

When a message is retrieved from MSMQ, the MSMQ-MQSeries Bridge will have converted the MSMQ PROPID_M_LABEL and PROPID_M_LABEL_LEN properties from the MQSeries MQMD.ApplIdentityData field value.

The MQMDE extension has the following GUID for MQMD version 2.

```
static const GUID sg_MSMQExtMQMDE =
{ 0x18ae68f5, 0x989b, 0x11d3,
```

```
    { 0x8d, 0xf9, 0x0, 0x0, 0xf8, 0x1a,  0xea, 0x1f }
};
```

This section contains:

- Transaction Support Using MSMQ-MQSeries Bridge
- Security Using MSMQ-MQSeries Bridge
- Troubleshooting the MSMQ-MQSeries Bridge Extensions

Transaction Support Using MSMQ-MQSeries Bridge

The MSMQ-MQSeries Bridge supports both MSMQ and MQSeries transactions. The procedure for sending a group of transacted messages is similar in either direction, from MSMQ to MQSeries or from MQSeries to MSMQ. Your application should follow these three basic procedures:

- Open a transaction
- Send the messages
- Commit the transaction

At this point, the group of messages reaches the MSMQ connector queue or the MQSeries transmission queue. Only then does the MSMQ-MQSeries Bridge transmit the messages to the other messaging system. If your application aborts the transaction instead of committing, MSMQ-MQSeries Bridge does not handle the messages at all.

Even after you commit a transaction, it is still possible that MSMQ-MQSeries Bridge cannot transmit all the messages. This may occur, for example, if some of the messages are addressed to queues that don't exist in the recipient messaging system. MSMQ-MQSeries Bridge places any undeliverable messages on its dead letter queue.

In the MSMQ-to-MQSeries direction, the MSMQ-MQSeries Bridge sends transacted messages by a transactional message pipe and untransacted messages by regular message pipe. In the MQSeries-to-MSMQ direction, MQS->MSMQ message pipe, and MQS->MSMQ transactional message pipe do not depend on transaction status.

Security Using MSMQ-MQSeries Bridge

Message authentication and message body encryption are supported from the MSMQ sending application up to the MSMQ-MQSeries Bridge. Authentication and message body encryption from the MSMQ-MQSeries Bridge to MQSeries, or from MQSeries to MSMQ, are not currently supported.

Troubleshooting the MSMQ-MQSeries Bridge Extensions

The Microsoft MSMQ-MQSeries Bridge generally ignores warnings that it receives from MSMQ or MQSeries, but errors are not ignored. Where possible, the MSMQ-MQSeries Bridge transmits messages despite any warnings.

If the MSMQ-MQSeries Bridge is unable to transmit a message to MSMQ or MQSeries, it places the message on one of its dead letter queues. This can happen, for example, if a message contains an unsupported MSMQ or MQSeries version identifier or if MSMQ-MQSeries Bridge encounters an error in the recipient messaging system.

The dead letter queues are MSMQ queues located on the MSMQ-MQSeries Bridge machine. There are two dead letter queues used for this purpose with the following names:

Dead Letter Queue Names	Comments
MQBridge dead letter	Used for untransacted messages when errors occur.
MQBridge xact dead letter	Used for transacted messages when errors occur.

Note that these are different from the MSMQ and MQSeries dead letter queues, where the messaging systems place expired or incorrectly addressed messages.

You can determine whether there are messages on the dead letter queues using the MSMQ-MQSeries Bridge Manager.

If the MSMQ-MQSeries Bridge cannot deliver a message to MQSeries, it records the error in the extension property PROPID_M_EXTENSION of the original MSMQ message and places the message on the dead letter queue. This extension property can be examined to determine the nature of the error encountered.

Registry Settings Used By MSMQ-MQSeries Bridge Extensions

The Microsoft MSMQ-MQSeries Bridge Extensions uses a number of registry settings for configuration and proper operation. The configuration registry settings are located under the **HKEY_LOCAL_MACHINE\Software\Microsoft\SNA Server\CurrentVersion\Setup** key. These registry settings include the following subkeys:

Subkey	Comment
RootDir	Stores the path to root directory where the Host Integration Server was installed. The system directory below this root directory is the location where the MSMQ-MQSeries Bridge Extensions DLL and other support DLLs are installed.

MSMQ-MQSeries Bridge Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide lists the extensions and components that make up the MSMQ-MQSeries bridge.

This section contains:

MSMQ-MQSeries Bridge Extensions Reference

This section provides an alphabetic reference to all of the API calls for the MSMQ-MQSeries Bridge Extension Property API.

This section contains:

EPAdd

The **EPAdd** function adds a new extension field at the end of an existing EP object and optionally returns a cursor pointing to the new extension field in the EP object.

```
HRESULT EPAdd(
    HANDLE hExtension,
    PCGUID pFieldID
    void *pFieldData,
    DWORD dwDataLength,
    PHANDLE phCursor
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle to the EP object that is to have data added.
pFieldID
Supplied parameter. A pointer to the GUID of the new extension field. A GUID is 16 bytes in length.
pFieldData
Supplied parameter. A pointer to the buffer containing the data for the new extension field.
dwDataLength
Supplied parameter. The length of the buffer containing the data for the new extension field.
phCursor
Supplied and returned parameter. A pointer to a cursor, which points to the new extension field. If *phCursor* is NULL when this function is called, the cursor is not created.

Return Codes

MQ_OK
The function executed successfully.
MQ_ERROR_ALLOC_FAIL
The function failed because memory could not be allocated for the internal data buffers used to extend the EP object.
MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.
MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.

Remarks

In an EP object possessing a cursor, the extension fields are sorted in ascending order based on the GUID of each extension field. The message extension API functions may run more slowly while the cursor is in effect.

All cursors are canceled if the EPDeleteAll function is called with a *pFieldID* of NULL.

The following example illustrates how to use this function.

```
HANDLE hExt;
HANDLE hCursor;
GUID guid;
...
/* Add a new field containing the data "test" */
EPAdd(hExt, &guid, "test", 5, NULL);

/* Add a new field and create a cursor */
EPAdd(hExt, &guid, "another test", 13, &hCursor);
```

See Also

EPDelete, EPDeleteAll

EPClose

The **EPClose** function closes an open EP object freeing the extension handle and associated memory of an EP object. The entire contents of the object are deleted.

```
HRESULT EPClose(
    PHANDLE phExtension
    );
```

Parameters

phExtension
Supplied and returned parameter. A pointer to an existing EP object handle to close.

Return Codes

MQ_OK
The function executed successfully.
MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.
MQ_ERROR_INVALID_PARAMETER
The function failed because the parameter passed to this function is invalid.

Remarks

If the EP object handle is successfully closed, *phExtension* is reset to NULL on output.

See Also

EPOpen

EPDelete

The **EPDelete** function deletes a single extension field from an existing EP object.

```
HRESULT EPDelete(
    HANDLE hExtension,
    PHANDLE phCursor
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle to the EP object that is to have extension field deleted.

phCursor
Supplied and returned parameter. On input, *phCursor* is a cursor pointing to the extension field to be deleted. On output, *phCursor* is set to the next extension field in the extension, or to NULL if there are no more fields.

Return Codes

MQ_OK
The function executed successfully.

MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.

MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.

Remarks

After successful deletion, the cursor is set to point to the next field after the deleted one. If the last field is deleted, the cursor is set to NULL (to the beginning).

In an EP object possessing a cursor, the extension fields are sorted in ascending order based on the GUID of each extension field. The message extension API functions may run more slowly while the cursor is in effect.

All cursors are canceled if the EPDeleteAll function is called with a *pFieldID* of NULL.

See Also

EPDeleteAll

EPDeleteAll

The **EPDeleteAll** function deletes all extension fields from an existing EP object or all extension fields matching a specific GUID.

```
HRESULT EPDeleteAll(
    HANDLE hExtension,
    PCGUID pFieldsId,
    PHANDLE phCursor
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle to the EP object that is to have extension field deleted.

pFieldsId
Supplied parameter. A pointer to a 16-byte buffer containing the GUID of the fields to delete. If this parameter is NULL, all extension fields are deleted.

phCursor
Supplied and returned parameter. On output, this field is a pointer to an extension field cursor. The cursor is positioned at the first field of the next higher GUID after the one that was deleted. If there are no more GUIDs or if all fields were deleted, *phCursor* is set to NULL. The *phCursor* may be NULL on input if no cursor is desired.

Return Codes

MQ_OK
The function executed successfully.

MQ_ERROR_EXTENSION_FIELD_NOT_FOUND
The function failed because the extension field matching the specified GUID could not be found.

MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.

MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.

Remarks

If the *phCursor* parameter is not NULL, **phCursor* will point to the next field after all the deleted ones (even if MQ_ERROR_EXTENSION_FIELD_NOT_FOUND is returned). If the last field is deleted, the cursor is set to NULL.

In an EP object possessing a cursor, the extension fields are sorted in ascending order based on the GUID of each extension field. The message extension API functions may run more slowly while the cursor is in effect.

All cursors are canceled if the EPDeleteAll function is called with a *pFieldID* of NULL.

The following example illustrates how to use this function.

```
HANDLE hExt;
GUID guid;
HANDLE hCursor;
...
/* Delete all fields having a specified GUID and set a cursor */
EPDeleteAll(hExt, &guid, &hCursor);

/* Delete all extension fields */
EPDeleteAll(hExt, NULL, NULL);
```

See Also

EPDelete

EPGet

The **EPGet** function reads a specified extension field from an EP object, storing the GUID, length, and data subfields in separate variables. **EPGet** can also be used to locate extension fields containing a specified GUID

```
HRESULT EPGet(
    HANDLE hExtension,
    NAVTYPE Directive,
    PHANDLE phCursor
    GUID *pFieldID,
    void *pFieldData,
    PDWORD pdwDataLength,
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle.

Directive
Supplied parameter. This parameter and *phCursor* together control the behavior of the function. Possible values and their usage are discussed in the table following this parameter list.

phCursor
Supplied and returned parameter. A pointer to a cursor, which points to the matching extension field. If *phCursor* is NULL, the first field having a GUID matching *pFieldID* is read, and the cursor is positioned to this field. See the *Directive* argument for specific details.

pFieldID
Supplied and returned parameter. If the *Directive* argument is EP_NEXT_KEY_FIELD, *pFieldId* is a pointer to a 16-byte buffer containing the GUID to be read. If the *Directive* argument is EP_CURRENT_FIELD or EP_NEXT_FIELD, on output this parameter is a pointer to a 16-byte buffer where the function stores the GUID. The GUID can be NULL if the GUID output is not desired.

pFieldData
Supplied and returned parameter. On input, a pointer to a buffer where the function should store the extension field data. On input, this parameter can be NULL if the data output is not desired. On output, a pointer to a buffer where the function stores the extension field data.

pdwDataLength
Supplied and returned parameter. On input, the length of the buffer for the data from the extension field. On output, the actual length of the data. If the buffer is too short or NULL, the data is not read but *pdwDataLength* is reset to the required buffer length.

Values for the *Directive* parameter

Directive	Description
EP_CURRENT_FIELD	Retrieves the extension field pointed to by the *phCursor* parameter, which must be a valid non-NULL cursor handle.
EP_NEXT_FIELD	Advances the cursor and reads the next field. If there are no more fields, returns MQ_ERROR_EXTENSION_FIELD_NOT_FOUND and *phCursor* is set to NULL. If *phCursor* is NULL or **phCursor* is NULL, the cursor is positioned to the first field (sorted in ascending order of GUID) and this field is read. If *phCursor* is not NULL, **phCursor* is set to the extension field. If *phCursor* is not NULL and **phCursor* is not NULL, the cursor (*phCursor*) is positioned to the first field (sorted in ascending order of GUID) and this field's ID and data are read and returned.
EP_NEXT_KEY_FIELD	Advances the cursor and reads the next field having a GUID matching *pFieldID*. If there are no more matching fields, returns MQ_ERROR_EXTENSION_FIELD_NOT_FOUND and sets *phCursor* to the first field having the next higher GUID, or to NULL if there are no more fields. If *phCursor* is NULL or **phCursor* is NULL, then the first field having a GUID matching *pFieldID* is read, and the cursor is positioned to this field. If the field is found it's field ID and data are returned and if *phCursor* is not NULL, **phCursor* is set to the field.

Return Codes

MQ_OK
The function executed successfully.
MQ_ERROR_ALLOC_FAIL
The function failed because memory could not be allocated for the internal data buffers used for the EP object.
MQ_ERROR_EXTENSION_FIELD_NOT_FOUND
The function failed because the extension field matching the specified GUID could not be found.
MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.
MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.
MQ_ERROR_USER_BUFFER_TOO_SMALL
The function failed because the length of the buffer passed was too small for the data.

Remarks

In an EP object possessing a cursor, the extension fields are sorted in ascending order based on the GUID of each extension field. The message extension API functions may run more slowly while the cursor is in effect.

All cursors are canceled if the EPDeleteAll function is called with a *pFieldID* of NULL.

The following example illustrates how to use this function.

```
HANDLE hExt;
HANDLE hCursor;
GUID guid;
void *pBuffer;
DWORD dwSize, dwCount;
DWORD dwTotalSize = 0;
...

/* Retrieve one field by GUID */
dwSize = 1024;
EPGet(hExt, EP_NEXT_KEY_FIELD, NULL, &guid, pBuffer, &dwSize);

/* Count the fields in a message extension */
for (hCursor = NULL, dwCount = 0;
     EPGet(hExt, EP_NEXT_FIELD, &hCursor, NULL, NULL, NULL)==MQ_OK;
     dwCount++);

/* Compute the total length of all extension fields
   having a given GUID */
for (hCursor = NULL, dwCount = 0;
          EPGet(hExt, EP_NEXT_KEY_FIELD, &hCursor, &guid, NULL,
              &dwSize) == MQ_OK;
     dwTotalSize += dwSize);
```

See Also

EPDelete, EPDeleteAll

EPGetBuffer

The **EPGetBuffer** function converts an EP object to a message extension (PROPID_M_EXTENSION format) that can be sent in a message and packs the message extension into the supplied buffer.

```
HRESULT EPGetBuffer(
    HANDLE hExtension,
    void *pBuf,
    PDWORD pdwBufLength,
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle.
pBuf
Supplied parameter. A pointer to the buffer where this function will store the PROPID_M_EXTENSION message extension.
pdwBufLength
Supplied and returned parameter. On input, the length of the buffer for the message extension. On output, the actual length of the stored data. If the buffer is too short or NULL, the data is not converted but *pdwBufLength* is reset to the required buffer length.

Return Codes

MQ_OK
The function executed successfully.
MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.
MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.
MQ_ERROR_USER_BUFFER_TOO_SMALL
The function failed because the length of the buffer passed was too small for the data.

Remarks

If this function executed successfully, the *pdwBufLength* parameter contains the actual length of the packed extension buffer. If MQ_ERROR_USER_BUFFER_TOO_SMALL is returned, *pdwBufLength* contains the required buffer length.

The following example illustrates how to use this function.

```
HANDLE hExt;
void *pBuffer;
DWORD dwSize;

/* Read the required buffer length */
dwSize = 0;
EPGetBuffer(hExt, NULL, &dwSize);

/* Allocate the buffer */
pBuffer = malloc(dwSize);

/* Write the message extension to the buffer */
EPGetBuffer(hExt, pBuffer, &dwSize);
```

See Also

EPAdd, EPOpen

EPOpen

The **EPOpen** function creates an EP object and optionally unpacks the supplied message extension buffer into it.

```
HRESULT EPOpen(
    PHANDLE phExtension,
    void *pExtBuffer,
    DWORD dwExtBufLength
    );
```

Parameters

phExtension
Returned parameter. A pointer to an EP object handle of the EP object that is created.

pExtBuffer
Supplied parameter. The pointer to a buffer containing the message extension data in the sequence GUID (16 bytes), length of data (4 bytes), data, GUID, length of data, data, etc.

dwExtBufLength
Supplied parameter. The length of the buffer containing the message extension data.

Return Codes

MQ_OK
The function executed successfully.

MQ_ERROR_ALLOC_FAIL
The function failed because memory could not be allocated for the EP object handle and internal data buffers.

MQ_ERROR_CORRUPTED_EXTENSION_BUFFER
The function failed because the buffer containing the message extension data was corrupted.

MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.

Remarks

If a NULL pointer is passed for *pExtBuffer* or *dwExtBufLength* is zero, an EP object with no extension fields is created.

The following example illustrates how to use this function.

```
HANDLE hExt1, hExt2;
void *pBuffer;
DWORD dwBufLength;
...
/* Create an empty EP object */
EPOpen(&hExt1, NULL, 0);
/* Create an EP object, copying data from a message extension */
EPOpen(&hExt2, pBuffer, dwBufLength);
```

See Also

EPClose

EPUpdate

The **EPUpdate** function updates (replaces) the data and length subfields of an existing extension field in an EP object.

```
HRESULT EPUpdate(
    HANDLE hExtension,
    HANDLE hCursor
    void *pFieldData,
    DWORD dwDataLength,
    );
```

Parameters

hExtension
Supplied parameter. The EP object handle to the EP object that is to have data updated.
hCursor
Supplied parameter. The cursor pointing to the extension field to be updated which must not be NULL.
pFieldData
Supplied parameter. A pointer to the buffer containing the new data for the extension field. This parameter may be NULL for an empty data subfield.
dwDataLength
Supplied parameter. The length of the buffer containing the new data for the extension field.

Return Codes

MQ_OK
The function executed successfully.
MQ_ERROR_ALLOC_FAIL
The function failed because memory could not be allocated for the internal data buffers used to update the EP object.
MQ_ERROR_INVALID_HANDLE
The function failed because the EP object handle passed to the function is invalid.
MQ_ERROR_INVALID_PARAMETER
The function failed because one or more of the parameters passed to this function are invalid.

Remarks

The following example illustrates how to use this function.

```
HANDLE hExt;
HANDLE hCursor;
GUID guid;
...
/* Find an extension field containing a specified GUID */
EPGet (hExt, EP_NEXT_KEY_FIELD, &hCursor, &guid, NULL, 0)
/* Change the length and data subfields of the extension field */
EPUpdate(hExt, hCursor, "newdata", 8);
```

See Also

EPAdd, EPGet

SDK Components for MSMQ-MQSeries Bridge Extensions

The Microsoft® Host Integration Server 2000 SDK contains software components used for application integration using messaging and the MSMQ-MQSeries Bridge. The components used for application integration using the MSMQ-MQSeries Bridge are described in the following topics.

This section contains:

Program and DLL Files for MSMQ-MQSeries Bridge

The following executable system files, DLL library files, and other files are included with the Host Integration Server 2000 SDK for use with the MSMQ-MQSeries Bridge:

File name	Description
BCluster.exe	A program used to create or remove the MSMQ-MQSeries Bridge Service resource in a cluster.
explres.dll	The resource file for Q2QEXPL.exe, the MSMQ-MQSeries Bridge explorer program.
MQBInst.dll	The COM component used to create, delete, or modify MSMQ-MQSeries Bridge objects in Active Directory.
mqfrgnky.dll	A library to provide a function for the MSMQ-MQSeries Bridge to store a public key in a foreign computer object (i.e. MQFrgn_StorePubKeysInDS).
MQSRRecv.exe	A sample program that uses the MQSeries API to receive messages from a specified MQSeries queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MQSRSend.exe	A sample program that uses the MQSeries API to send 10 test messages to a specified MQSeries queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MSMQRecv.exe	A sample program that uses the MSMQ API to receive messages from a specified MSMQ queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MSMQSend.exe	A sample program that uses the MSMQ API to sends 10 test messages to a specified MSMQ local or foreign queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
Q2QCLDLL.dll	A helper DLL (now obsolete) that contains functions to work with a cluster.
Q2QEXPL.exe	The MSMQ-MQSeries Bridge explorer program.
Q2QGW.exe	The MSMQ-MQSeries Bridge service program.
q2qmsg.dll	The Event Log message file.
q2qperf.ini	The performance counter definition file for the MSMQ-MQSeries Bridge.
q2qprfdl.dll	The performance counter implementation DLL for the MSMQ-MQSeries Bridge extension.
q2qprfsm.def	Defines the performance counter object in q2qperf.ini.
Q2QSHDLL.dll	A helper DLL (now obsolete) that contains functions for installing and uninstalling the MSMQ-MQSeries Bridge.
SHDLLRes.dll	The resource DLL (now obsolete) for Q2QSHDLL.DLL for installing and uninstalling the MSMQ-MQSeries Bridge.
wmiMQBridge.dll	The MSMQ-MQSeries Bridge WMI Provider.
wmimqbridge.mof	The WMI Managed Object File (MOF) for the MSMQ-MQSeries Bridge.

The following executable system files, DLL library files, and other files are included with SNA Server 4.0 Service Pack 4 for use with the MSMQ-MQSeries Bridge:

File name	Description
EPRECV.EXE	A sample program that uses the MSMQ-MQSeries Bridge Extensions API to display the MQMD structure in the MSMQ extension property.
EPSEND.EXE	A sample program uses the MSMQ-MQSeries Bridge Extensions API to override the default MSMQ-MQseries Bridge message property mapping MsgType, ReplyToQMgr, and ReplyToQ in the MQSeries MQMD structure.
explres.dll	The resource file for Q2QExpl.exe, the MSMQ-MQSeries Bridge explorer program.
Inetwh32.dll	A support DLL for online help.
MQBridge.chm	The MSMQ-MQSeries Bridge online help file.

MQSRRecv.exe	A sample program that uses the MQSeries API to receive messages from a specified MQSeries queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MQSRSend.exe	A sample program that uses the MQSeries API to send 10 test messages to a specified MQSeries queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MSMQRecv.exe	A sample program that uses the MSMQ API to receive messages from a specified MSMQ queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
MSMQSend.exe	A sample program that uses the MSMQ API to sends 10 test messages to a specified MSMQ local or foreign queue. This program can be used to test the operation of the MSMQ-MQSeries Bridge.
q2qcldll.dll	A helper DLL that contains functions to work with a cluster.
Q2QCIlns.exe	A helper program for the MSMQ-MQSeries Bridge cluster installation.
Q2QClUni.exe	A helper program for the MSMQ-MQSeries Bridge cluster uninstallation.
Q2QExpl.exe	The MSMQ-MQSeries Bridge explorer program.
q2qgw.exe	The MSMQ-MQSeries Bridge service program.
Q2Qgwy.CNT	The online help index file for the MSMQ-MQSeries Bridge service program.
Q2Qgwy.hlp	The online help file for the MSMQ-MQSeries Bridge service program.
Q2QInsSt.exe	A program used to launch Q2QCIlIns.exe, the MSMQ-MQSeries Bridge cluster installation program.
Q2QMSG.dll	The Event Log message file.
Q2qperf.ini	The performance counter definition file for the MSMQ-MQSeries Bridge.
q2qprfdl.dll	The performance counter implementation DLL for the MSMQ-MQSeries Bridge extension.
Q2QSHDLL.dll	A helper DLL (now obsolete) that contains functions for installing and uninstalling the MSMQ-MQSeries Bridge.
SHDLLRes.dll	The resource DLL (now obsolete) for Q2QSHDLL.DLL for installing and uninstalling the MSMQ-MQSeries Bridge.
SNAVER.exe	A utility program to retrieve the SNA Server program version.

Symbol Files for MSMQ-MQSeries Bridge

The following symbol files for use when debugging are included with Host Integration Server 2000 for use with the MSMQ-MQSeries Bridge. These files are installed as part of the Host Integration Server package and a copy of these files are also located on the Host Integration Server CD-ROM under the Support\Symbols folder:

File name	Description
EXE\BCluster.dbg	Symbols from BCluster.exe
DLL\explres.dbg	Symbols from Explres.dll
DLL\MQBInst.dbg	Symbols from MQBInst.dll
EXE\MQSRRecv.dbg	Symbols from MQSRRecv.exe
EXE\MQSRSend.dbg	Symbols from MQSRSend.exe
EXE\MSMQRecv.dbg	Symbols from MSMQRecv.exe
EXE\MSMQSend.dbg	Symbols from MSMQSend.exe
DLL\Q2QCLDLL.dbg	Symbols from Q2QCLDLL.dll.
EXE\Q2QCIIns.dbg	Symbols from Q2QCIIns.exe.
EXE\Q2QClUni.dbg	Symbols from Q2QClUni.exe.
EXE\Q2QEXPL.dbg	Symbols from Q2QEXPL.exe.
EXE\Q2QGW.dbg	Symbols from Q2QGW.exe.
EXE\Q2QInst.dbg	Symbols from Q2QInst.exe.
DLL\q2qmsg.dbg	Symbols from q2qmsg.dll.
DLL\q2qprfdl.dbg	Symbols from q2qprfdl.dll.
DLL\Q2QSHDLL.dbg	Symbols from Q2QSHDLL.dll.
DLL\SHDLLRes.dbg	Symbols from SHDLLRes.dll.
DLL\wmiMQBridge.dbg	Symbols from wmiMQBridge.dll.

The following symbol files for use when debugging are included with SNA Server 4.0 Service Pack 4 for use with the MSMQ-MQSeries Bridge. These files are located on the SNA Server CD-ROM under the mqbridge folder:

File name	Description
EPRECV.DBG	Symbols from EPRECV.EXE
EPSEND.DBG	Symbols from EPSEND.EXE
eplres.dbg	Symbols from explres.dll
MQSRRecv.dbg	Symbols from MQSRRecv.exe
MQSRSend.dbg	Symbols from MQSRSend.exe
MSMQRecv.dbg	Symbols from MSMQRecv.exe
MSMQSend.dbg	Symbols from MSMQSend.exe
Q2QCLDLL.dbg	Symbols from q2qcldll.dll
Q2QCIIns.dbg	Symbols from Q2QCIIns.exe
Q2QClUni.dbg	Symbols from Q2QClUni.exe
Q2QEXPL.dbg	Symbols from Q2QExpl.exe
Q2QGW.dbg	Symbols from q2qgw.exe
Q2QInsSt.dbg	Symbols from Q2QInsSt.exe
q2qmsg.dbg	Symbols from Q2QMSG.dll
q2qprfdl.dbg	Symbols from q2qprfdl.dll
Q2QSHDLL.dbg	Symbols from Q2QSHDLL.dll
SHDLLRes.dbg	Symbols from SHDLLRes.dll
SNAVER.dbg	Symbols from SNAVER.EXE

Header Files for MSMQ-MQSeries Bridge

Provider-specific header files needed to build the MSMQ-MQSeries Bridge sample applications are included with Host Integration Server 2000. These files are installed as part of the Host Integration Server package and a copy of these files are also located on the Host Integration Server CD-ROM under the SDK\Include folder:

The following provider-specific files are provided with Host Integration Server for developing applications using the MSMQ-MQSeries Bridge:

File name	Description
msmqep.h	GUID definitions, enumeration constants, and error codes for use with the MSMQ-MQSeries Bridge.

Provider-specific header files needed to build the MSMQ-MQSeries Bridge sample applications are included with SNA Server 4.0 Service Pack 4. These files are installed as part of the MSMQ-MQSeries Bridge with SNA Server and a copy of these files are also located on the SNA Server 4.0 CD-ROM under the mqbridge folder:

The following provider-specific files are provided with SNA Server 4.0 Service Pack 4 for developing applications using the MSMQ-MQSeries Bridge:

File name	Description
mqsrext.h	GUID definitions, enumeration constants, and error codes for use with the MSMQ-MQSeries Bridge.
MSMQExt.h	Function prototypes and enumeration constants for use with MSMQ-MQSeries Bridge extended functions.
Q2qprfsm.h	Definitions for the Performance counter object in q2qperf.ini.

Import Library Files for MSMQ-MQSeries Bridge

Provider-specific import library files needed to build the MSMQ-MQSeries Bridge extension sample applications are included with Host Integration Server 2000. These files are installed as part of the Host Integration Server package and a copy of these files are also located on the Host Integration Server CD-ROM under the SDK\Lib folder:

The following provider-specific files are provided with Host Integration Server for developing applications using the MSMQ-MQSeries Bridge:

File name	Description
msmqep.lib	Import library of functions for use with the MSMQ-MQSeries Bridge extension.

Provider-specific import library files needed to build the MSMQ-MQSeries Bridge extension sample applications are included with SNA Server 4.0 Service Pack 4. These files are installed as part of the MSMQ-MQSeries Bridge with SNA Server and a copy of these files are also located on the SNA Server CD-ROM under the mqbridge folder:

The following provider-specific files are provided with SNA Server 4.0 Service Pack 4 for developing applications using the MSMQ-MQSeries Bridge:

File name	Description
MSMQEP.lib	Import library of functions for use with the MSMQ-MQSeries Bridge extensions.

Application Integration Samples

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about the sample applications that implement the MSMQ-MQSeries bridge and the COM Transaction Integrator (COMTI).

This section contains:

Sample Programs for MSMQ-MQSeries Bridge

The source code for several sample programs that illustrate using MSMQ-MQSeries Bridge are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\Bridge subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\Bridge subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\Bridge subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files in the following subdirectories:

Subdirectory	Description
EPRecv	A sample program in C that uses the MSMQ-MQSeries Bridge Extensions API to display the MQMD structure in the MSMQ extension property.
EPSend	A sample program in C that uses the MSMQ-MQSeries Bridge Extensions API to override the default MSMQ-MQseries Bridge message property mapping MsgType, ReplyToQMgr, and ReplyToQ in the MQSeries MQMD structure.
MQSRRecv	A sample program in C that uses the MQSeries API to receive messages from a specified MQSeries queue.
MQSRSend	A sample program in C that uses the MQSeries API to send 10 test messages to a specified MQSeries queue.
MSMQRecv	A sample program in C that uses the MSMQ API to receive messages from a specified MSMQ queue.
MSMQSend	A sample program in C that uses the MSMQ API to sends 10 test messages to a specified MSMQ local or foreign queue.
WMI	A collection of WMI sample scripts written in Microsoft® Active Server Pages (ASP) that show how to use WMI to configure the MSMQ-MQSeries Bridge.

Several sample programs with source code are provided with Host Integration Server 2000 that illustrate how to use the MSMQ-MQSeries Bridge and Bridge Extensions.

The MSMQ-MQSeries Bridge samples are designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio .NET interactive development environment (IDE). Most of these samples also require that the IBM MQSeries Client toolkit has been installed, providing access to several MQSeries include and lib files.

To build the MSMQ-MQSeries Bridge samples using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory(by default, C:\Program Files\Microsoft Visual Studio\VC98\Bin for VS6 or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools for VS.NET))
- Set the MQS_INC environment variable so it points to the INCLUDE directory where MQSeries was installed. The default location for this variable is normally C:\Program Files\MQSeries Client\tools\c\include.
- Set the MQS_LIB environment variable so it points to the LIB directory where MQSeries was installed. The default location for this variable is normally C:\Program Files\MQSeries Client\tools\lib.

For example, set the following environment variables for building the MQS samples:

```
set MQS_INC=C:\Program Files\MQSeries Client\tools\c\include
set MQS_LIB=C:\Program Files\MQSeries Client\tools\lib
```

To build all the C/C++ samples (EPSend, EPRecv, MQSRRecv, MQSRSend, MSMQSend and MSMQRecv), open an MS-DOS Command Prompt window, navigate to Bridge subdirectory, and invoke NMAKE. This will recursively invoke NMAKE and build all of the Bridge samples.

To build a specific sample (EPSend, for example) using the command-line compiler, open an MS-DOS Command Prompt window, navigate to the appropriate subdirectory (Bridge\EPSend, for example), and invoke NMAKE.

To build a specific sample (EPSend, for example) using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (epsend.vcproj, for example) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build. Note that several of the MSMQ-MQSeries Bridge samples require access to the IBM MQSeries Client toolkit include and lib files.

The VC7 project files for these samples assume that the IBM MQSeries Client toolkit is installed in the default location at C:\Program Files\MQSeries Client\tools. If the IBM MQSeries Client toolkit is installed in a different location, the VC7 project files will need to be modified. For each C source file, the **Additional Include Directories** property under **C/C++/General** will need to be changed. For the target, the **Additional Dependencies** property under **Linker/Input** will need to be changed.

This section contains:

- EPRecv sample program
- EPSend sample program
- MQSRRecv sample program
- MQSRSend sample program
- MSMQRecv sample program
- MSMQSend sample program.
- WMI MSMQ-MQSeries Bridge sample programs

EPRecv Sample

The MSMQ-MQSeries Bridge Extension API can be used to obtain the original MQSeries message properties for an MQSeries message sent using the MSMQ-MQSeries Bridge to MSMQ. The Bridge\EPRecv folder contains a sample program written in C that receives messages from an MSMQ queue using the MSMQ APIs and prints the original MQSeries message MQMD properties in the PROPID_M_EXTENSION if they exist. The sample illustrates how to use MSMQ-MQSeries Bridge Extensions and can be used for testing or troubleshooting the MSMQ-MQSeries Bridge and Bridge Extensions.

The usage for this command-line tool is as follows:

```
EPRecv <computer name>\<queue name>
```

The only parameter. <computer name>\<queue name> is the path name of the specified MSMQ queue name where messages are received. This MSMQ queue name can be specified in UNC or DNS format.

Sample program usage and sample output are listed below.

```
eprecv MSBRIDGE\QUEUE

Queue opened.
Waiting for messages to arrive.
Use CTLR-C to stop.
-------> Message arrived:
Label = ''
Body (256) = 'Test Message 0 - 19:41:26'
Body Type (4113)
Extension property found.  Dumping values:
    MQMD1 Extension Field found.  Dumping values:
        MQMD1.Report = 00000000   MQMD1.MsgType = 00000008
        MQMD1.Feedback = 00000000   MQMD1.Priority = 0
        MQMD1.Version = 00000001  MQMD1.Expiry = -1
        MQMD1.ReplyToQMgr = 'BRIDGE2K_QM                                    '
        MQMD1.ReplyToQ    = '                                               '
        MQMD1.UserIdentifier = 'testuser     '
        MQMD1.ApplIdentityData = '            '
        MQMD1.PutApplName = 'n Server\sys'
        MQMD1.PutDate = '20000628'
        MQMD1.PutTime = '02530720'
        MQMD1.MsgId = '414D512053544152 5741525F514D2020 9197523913300000'
        MQMD1.CorrelId = '0000000000000000 0000000000000000 0000000000000000'
```

EPSend Sample

The MSMQ-MQSeries Bridge Extension API can be used to override the default MSMQ-MQSeries Bridge message property mapping. The Bridge\EPSend folder contains a sample written in C that illustrates how to use the MSMQ-MQSeries Bridge Extension API to override the default MSMQ-MQSeries Bridge message propery mapping for MsgType, ReplyToQMgr, and ReplyToQ in the MQSeries MQMD structure. This sample sends messages to MSMQ queue, overiding the default values for these MQMD extension fields. The sample illustrates how to use MSMQ-MQSeries Bridge Extensions and can be used for testing or troubleshooting the MSMQ-MQSeries Bridge and Bridge Extensions.

The usage for this command-line tool is as follows:

```
EPSend <computer name>\<queue name>
```

The only parameter <computer name>\<queue name> is the path name of the specified MSMQ queue name where messages are to be sent. This MSMQ queue name can be specified in UNC or DNS format.

Sample program usage and sample output are listed below.

```
epsend IBMNT_QM\QUEUE

Queue opened.
Reply Q Manager Name : MSBRIDGE1
Reply Q Name : QUEUE
------> Sending message (Use CTRL-C to stop).Label: ABC
Body: ABC
```

MQSRRecv Sample

The Bridge\MQSRRecv folder contains a sample program written in C that uses the MQSeries API to receive messages from a specified MQSeries queue. This sample can be used to receive messages sent from MQSeries or sent from MSMQ using the MSMQ-MQSeries Bridge. The sample can be used for testing or troubleshooting the MSMQ-MQSeries Bridge.

The usage for this command-line tool is as follows:

```
MQSRRecv <QM name> <queue name>
```

The first parameter, QM name, is the name of the MQSeries Queue Manager. The second parameter, queue name, is the queue name from which to receive the messages. Note that the program assumes that queue name is located on the QM name computer.

You can run the MQSRRecv program on a computer where the MQSeries Client is installed and configured. The environment variables used by the MQSeries client should point to the appropriate channel table file. The computer running the MSMQ-MQSeries Bridge is a good choice because it should already be properly configured.

MQSRSend Sample

The Bridge\MQSRSend folder contains a sample written in C that uses the MQSeries API to send 10 test messages to a specified MQSeries queue. This sample can be used to send messages to a specified MQSeries queue or to a specified MSMQ queue using the MSMQ-MQSeries Bridge. The sample can be used for testing or troubleshooting the MSMQ-MQSeries Bridge.

The usage for this command-line tool is as follows:

```
MQSRSend <local QM name> <destination QM name>
        <queue name>
```

The first parameter, local QM name, is the name of the immediate MQSeries Queue Manager to connect to (the server side of the MQI channel). MQSRSend needs this information to establish the MQI channel connection.

The second parameter, the destination QM name, is the destination queue manager for the messages. To send messages to MSMQ, specify the queue manager alias for the destination QM name representing the MSMQ queue.

The third parameter, queue name, is the name of the queue where the messages should be sent.

You can send MQSeries message to an MSMQ queue with one of the following methods.

1. Specify the MSMQ-MQSeries Bridge machine name in the destination QM name and the MSMQ format name in the queue name.
2. Define QREMOTE for the MSMQ destination QM name in MQSeries.

You can run the MQSRSend program on a computer where the MQSeries Client is installed and configured. The environment variables used by the MQSeries client should point to the appropriate channel table file. The computer running the MSMQ-MQSeries Bridge is a good choice because it should already be properly configured.

MSMQRecv Sample

The Bridge\MSMQRecv folder contains a sample written in C that uses the MSMQ API to receive messages from a specified MSMQ queue. This sample can be used to receive messages sent from MSMQ or receive messages sent from MQSeries using the MSMQ-MQSeries Bridge. The sample illustrates how to receive messages using MSMQ and can be used for testing or troubleshooting the MSMQ-MQSeries Bridge.

The usage for this command-line tool is as follows:

```
MSMQRecv <computer name>\<queue name>
```

The only parameter. <computer name>\<queue name> is the path name of the specified MSMQ queue name where messages are received. This MSMQ queue name can be specified in UNC or DNS format.

You can run the MSMQRecv program on any computer where MSMQ is installed, not necessarily the computer running the MSMQ-MQSeries Bridge.

MSMQSend Sample

The SDK\Samples\Bridge\MSMQSend folder contains a sample written in C that sends 10 test messages using the MSMQ APIs. This sample can be used to send messages to a specified MSMQ queue or a foreign MQSeries queue through the MSMQ-MQSeries Bridge. The sample illustrates how to send messages using MSMQ and can be used for testing or troubleshooting the MSMQ-MQSeries Bridge.

The usage for this command-line tool is as follows:

```
MSMQSend <computer name>\<queue name>
```

The only parameter. <computer name>\<queue name> is the path name of the specified MSMQ queue name where messages are to be sent. This MSMQ queue name can be specified in UNC or DNS format.

You can run the MSMQSend program on any computer where MSMQ is installed, not necessarily the computer running the MSMQ-MQSeries Bridge.

WMI MSMQ-MQSeries Bridge Sample

The Bridge\WMI folder contains a collection of Active Server Pages (ASP) for use with a Web server application that allow you to view and make changes to the MSMQ-MQSeries Bridge configuration using WMI. These sample applications require Microsoft Internet Information Server version 3.0 or higher with Active Server Pages be installed. Host Integration Server 2000 and Internet Information Server must be installed and running on the same computer.

The WMI ASP samples must be installed into the Web server's public directories below WWWRoot. Copy the contents of the entire WMI directory from the SDK\Samples\Bridge\WMI subdirectory to your WWWROOT directory on the Web server. After these files have been copied you should have a WWWROOT\WMI folder containing a number of ASP and GIF files.

The samples may then be run by opening Internet Explorer or some other Web browser on the same computer or a different computer and entering the following URL in the address line:

```
http://<computer name>/WMI/WMI_Test_Main.asp
```

Substitute the network name of the computer hosting the Web server and the MSMQ-MQSeries Bridge for the computer name (in angle brackets in the URL above). This will open the main page of the Bridge WMI ASP application and allow you to select any of the other sample ASP pages. Information about each sample is provided on this web page.

These ASP pages illustrate using WMI to view and make changes to the MSMQ-MQSeries Bridge configuration. The management functions supported by this application allow you to create a new instance, move to other instances (previous and next), delete an instance, and save an instance.

The WMI subdirectory below WWWROOT needs to have IIS security enabled (no anonymous access). Otherwise the scripts in these subdirectories will fail since the anonymous user account by default does not have access rights that would allow it to start or stop services on Windows NT or Windows 2000 or make changes to the MSMQ-MQSeries Bridge on the Host Integration Server system.

It is possible to host these ASP pages on a computer running the Web server that is different from the computer running the MSMQ-MQSeries Bridge and Host Integration Server. However, this requires some changes to the ASP pages to handle connections to a different computer, security, and authentication issues.

Sample Programs for COMTI

The source code for several sample programs that illustrate using features of the COM Transaction Integrator (COMTI) for CICS and IMS is included on the Microsoft® Host Integration Server 2000 CD-ROM as a part of the Host Integration Server Software Development Kit (SDK). COMTI allows developers to integrate component-based Windows® applications using COM, distributed COM, and COM+ with CICS and IMS transactions on IBM mainframes.

In addition to the COMTI samples included in the Host Integration Server SDK, there is a basic COMTI sample entitled CedarBank that is installed with COMTI when the COMTI feature option is selected during setup. The CedarBank sample is installed under the system\Tutorials\CedarBank subdirectory below where Host Integration Server is installed (the default location is C:\Program Files\Host Integration Server\system\Tutorials\CedarBank).

Note that documentation on COMTI is not included with the Host Integration Server SDK. Documentation on COMTI is included under Application Integration Services as part of the Host Integration Server 2000 user documentation. The documentation is also available in printable format on the Host Integration Server CD-ROM under the Documentation\Printable Books folder in the Application Integration Services.pdf file.

The COMTI sample programs are located in the \SDK\Samples\COMTI subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\COMTI subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN® Platform SDK, these samples are located under the Samples\NetDS\HIS\COMTI subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files in the following subdirectories:

Subdirectory	Description
BoundedRecordsets\COBOL-CICS	A sample program in COBOL using COMTI that illustrates the use of bounded recordsets. This subdirectory also contains a sample TLB file created using the COMTI Component Builder for this COBOL sample.
BoundedRecordsets\VB	A sample class defined in Microsoft Visual Basic® using COMTI that illustrates the use of bounded recordsets.
ProgrammingSpecifics	This folder contains a comprehensive sample that include Visual Basic client code as well as mainframe COBOL code and sample COMTI type libraries. This sample is intended to be a complete end-to-end sample demonstrating features of COMTI.
ProgrammingSpecifics\CICSNonlink	A sample program in COBOL using COMTI that demonstrates how to receive a COMTI fixed-sized data area greater than 32767. This sample is not intended to be a complete end-to-end program, but it demonstrates the receiving-side logic of a CICS Non-Link server application program using COMTI.
ProgrammingSpecifics\TCP	A set of several sample programs in COBOL using COMTI that demonstrate how to use a CICS TCP server application.
SyncLev2	A sample program in COBOL using COMTI that demonstrates how to use Sync Level 2.

These samples primarily use a remote environment of CICS using LU 6.2. These COMTI samples are designed to assist developers in creating code for specific COMTI features.

In order to first start working with COMTI, it is recommended that developers use the CedarBank tutorial that comes with the COMTI installation. The CedarBank tutorial illustrates how to use of all of the COMTI Remote Environments and includes the COMTI Type libraries and the COBOL code for the mainframe for all of the environments (IMS, CICS, APPC and TCP/IP). The CedarBank sample also includes sample programs for the client-side code written in Microsoft Visual Basic and Microsoft Visual C++®.

Once connectivity has been established by working with the CedarBank tutorial, then the COMTI samples included with the Host Integration Server SDK can be used to gain an understanding of more advanced COMTI features not covered by the CedarBank tutorial.

This section contains:

- Sync Level 2 Sample

Bounded Recordsets Sample

The COM Transaction Integrator (COMTI) for CICS and IMS can be used with Microsoft Visual Basic bounded recordsets. This sample includes Visual Basic code and CICS COBOL code showing how to use bounded recordsets by calling into a CICS transaction program via LU 6.2 (Remote Environment CICS using LU 6.2).

The Visual Basic code is in the COMTI\BoundedRecorsdsets\VB folder and defines a Visual Basic class file that illustrates the use of bounded recordsets. Note that additional Visual Code would need to be written to use this Visual Basic class file in a project. The Visual Basic code in the class file demonstrates how to create a recordset and populate it with data to send to the mainframe. Note that there is no code that actually displays the data that comes back from the mainframe. A developer can put in a breakpoint in the Visual Basic code using the debugger and use the immediate window to look at the data or insert further code to examine the data that is returned.

In the COMTI\BoundedRecorsdsets\COBOL-CICS folder, there is a COMTI type library (TLB file) that can be used with this sample. The type library is set up for accessing a transaction named GETI on the host. There is also sample COBOL code that can be compiled and linked on the mainframe side. The compiled code should be set up to run on the host as a transaction named GETI or the COMTI type library must be changed to reflect the name of the transaction if it is different.

Programming Specifics Sample

The COM Transaction Integrator (COMTI) for CICS and IMS supports a number of powerful features that are illustrated by this sample. In the COMTI\ProgrammingSpecifics folder, there is a complete Microsoft Visual Basic project that demonstrates the following features of COMTI:

- Returning a Recordset
- Variable Length Tables
- Handling REDEFINES Clauses
- Variably Sized Strings
- Handling FILLER
- Unbounded Recordsets
- In/Out Variable Length Table

The Visual Basic project contains comments with the Visual Basic source code that indicates which COBOL (*.cbl) file contains the associated COBOL code for the mainframe side. The sample type library is included for CICS using LU 6.2. There are seven methods defined in the type library. Check on the properties for each method and look at the Host Names tab to see what the Mainframe TP name is. The value of the mainframe TP name property can be changed to the name used when compiling the sample COBOL programs.

Programming Specifics CICSNonlink Sample

In the COMTI\ProgrammingSpecifics\CICSNonlink folder there is sample COBOL code showing how to receive more than 32K bytes of data in a single method call. This sample includes only the mainframe code (COBOL), and does not include the corresponding Visual Basic or Visual C++ code for the PC side. It is intended to demonstrate the receiving side logic of a COMTI Non-link server application. This COBOL program contains comments explaining what is being done in the code.

Programming Specifics TCP Sample

In the COMTI\ProgrammingSpecifics\TCP folder, there are sample COBOL Child Server programs that can be used for TCP/IP connections. The Cicscs.cbl code is a sample program for TCP using CICS with Concurrent Server (analogous to CICS using LU 6.2). The Mscmtics.clb code is a sample program for CICS calling a Link-to program (using CICS DPL). The Imsexpl.cbl code is a sample program for using IMS in the Explicit mode. The Imsimpl.cbl code is a sample program for using IMS in the Implicit mode. There are similar sample programs included with the CedarBank tutorial which directly reflect the CedarBank data being passed.

Sync Level 2 Sample

The COM Transaction Integrator (COMTI) for CICS and IMS supports the use of Sync Level 2 transactions. This sample includes COBOL source code illustrating transactional support (Sync Level 2) on the mainframe with CICS using LU 6.2. This sample only includes COBOL source code which contains comments describing each of the code sections. The sample code demonstrates executing a Commit and identifying that a Rollback has been requested from COMTI. Please note that there is also related documentation in Knowledge Base article Q220967 available at http://go.microsoft.com/fwlink/?LinkId=14803. This article explains COMTI Metadata elements so that developers can better understand how to use Metadata to allow the COBOL program to initiate a Rollback of a transaction.

Data Integration

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information required to develop applications to access data in an environment using Microsoft Host Integration Server 2000. This section provides documentation for developers on data access, data replication, and data tools.

This section contains:

Introduction to Data Integration

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop applications to access data in an environment using Microsoft® Host Integration Server 2000. This section provides documentation for developers on data access, data replication, and data tools.

Applications for data integration used in a Host Integration Server 2000 environment can be developed using several different development tools and application programming interfaces including:

- C or C++ applications that use OLE DB to access AS/400 and VSAM files.
- C or C++ applications that use OLE DB to access IBM Data Base 2 (DB2).
- C, C++, or Microsoft Visual Basic® applications that use Open Database Connectivity (ODBC) drivers to access IBM DB2.
- Microsoft Visual Basic applications that use ActiveX® Data Objects (ADO) to access AS/400 and VSAM files.
- Microsoft Visual Basic applications that use ActiveX Data Objects (ADO) to access IBM DB2 using OLE DB.
- Microsoft Visual Basic applications that use ActiveX Data Objects (ADO) to access IBM DB2 using ODBC.
- C, C++, or Microsoft Visual Basic applications that use the Host File Transfer ActiveX control to transfer files to and from MVS, OS/390, AS/400, and AS/36.
- C, C++, or Microsoft Visual Basic applications that use the Data Queue ActiveX control to access AS/400 data queues.

To use this guide effectively, you should be familiar with:

- Microsoft Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows® 2000
    - Microsoft Windows NT®
    - Microsoft Windows 98
    - Microsoft Windows 95
- SNA concepts

Depending on the application programming interface and development tools used, you should be familiar with:

- Microsoft COM objects
- Microsoft OLE DB
- Microsoft ADO
- Microsoft ODBC

This section contains:

Additional Resources

This guide does not describe the products, architectures, or standards developed by other companies or organizations. For information about Microsoft Windows NT and other operating systems, consult your product documentation.

For information about SNA architecture, refer to your system network documentation.

The following documents provide additional information about the OLE DB application programming interfaces (APIs):

- Microsoft Data Access Components (MDAC) Software Development Kit 2.5

The following documents provide additional information about ActiveX Data Objects:

- Microsoft Data Access Components (MDAC) Software Development Kit 2.5

The following documents and publications provide additional information about the Open Database Connectivity (ODBC) standard and ODBC programming:

- *Microsoft Data Access Components (MDAC) Software Development Kit 2.5*
- *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*
- *Inside ODBC*, written by Kyle Geiger and published by Microsoft Press

For more information about SNA and the Distributed Data Manager (DDM), see the following manuals:

- *IBM Distributed Data Management Architecture: General Information* (Document Number GC219527-3)
- *IBM OS400 Distributed Data Manager User's Guide*
- *IBM OS400 Distributed Data Manager Programmer's Guide*
- *IBM Systems Network Architecture: Technical Overview*
- *IBM Systems Network Architecture: Concepts and Products*
- *IBM SNA Format and Protocol Reference Manual: Architectural Logic*
- *IBM DFSMS/MVS Version 1 Release 2 DFM/MVS Guide and Reference* (Document Number SC26-4915-00)
- *IBM DFSMS/MVS Version 1 Release 3 DFM/MVS Guide and Reference* (Document Number SC26-4915-01)
- IBM DFSMS/MVS Version 1 Release 4 DFM/MVS Guide and Reference (Document Number SC26-4915-02)

For more information about IBM Data Base 2 (DB2), see the following manuals:

- *IBM DB2 for OS/390 Version 5 Reference for Remote DRDA: Requesters and Servers* (Document Number SC26-8964-00*)*
- *IBM DB2 for OS/390 Version 5 Application Programming and SQL Guide* (Document Number SC26-8958-00)
- *IBM DATABASE 2 Administration Guide for Common Servers Reference* (Document Number S20H-4580)
- *IBM DATABASE 2 Application Programming Guide for Common Servers Reference* (Document Number S20H-4643*)*
- *IBM DB2 Universal Database API Reference* (Document Number S10J-8167)
- IBM DB2 Universal Database Building Applications for Windows and OS/2 Environments Reference (Document Number S10J-8160)

For background information about logical unit (LU) 6.2, Advanced Program-to-Program Communications (APPC), or the Common Programming Interface for Communications (CPI-C), see the following manuals:

- *IBM SNA: Technical Overview*
- *IBM SNA: Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM SNA: Formats*
- *IBM Systems Network Architecture: Introduction to APPC*
- IBM Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2

OLE DB Providers

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information on using the OLE DB providers for AS/400, VSAM, and DB2.

This section contains:

Using the OLE DB Provider for AS/400 and VSAM

The Microsoft® OLE DB Provider for AS/400 and VSAM enables you to directly access record-level data in mainframe VSAM, Partitioned Data Sets (PDS), and midrange OS/400 files from within an OLE-aware application. The object linking and embedding database (OLE DB) is a standard set of interfaces that provide heterogeneous access to disparate sources of information located anywhere—file systems, e-mail folders, and databases. The OLE DB Provider for AS/400 and VSAM combines the universal data access of OLE DB with the record-level input/output (RLIO) protocol of IBM's Distributed Data Management (DDM) architecture.

DDM is a set of rules for distributing or extending the data management from one computer to another, such as from a mainframe to an AS/400 computer, or from one of these host computers to a server computer. By combining the OLE DB and DDM architectures, Microsoft enables organizations to preserve their investments in existing data management infrastructure, while extending universal data access to all enterprise-wide data sources.

This section contains:

Goals of the OLE DB Provider for AS/400 and VSAM

For the majority of enterprises today, the bulk of mission-critical information resides on IBM mainframe and AS/400 computers. This information is stored in records on the OS/400 and VSAM file systems. This information is created, owned, and often accessible by only host-based applications. In the mainframe world, these applications include CICS and DB2; other commercial applications; and a large number of custom applications written in COBOL, PL/I, and other languages. In the AS/400 world, these applications include primarily DB2 and commercial applications, plus a large number of custom RPG applications. A key point to make is that not all of these data sources are SQL-accessible. Many of the host data stores contain non-SQL-accessible data that is owned by something other than a traditional relational database management system (RDBMS).

These same enterprises rely on vast networks of personal computers to enable their users to achieve business goals. End users invariably rely on network e-mail, Microsoft® Windows® productivity applications such as Microsoft Office, and personal database programs such as Microsoft Access, to accomplish their daily tasks. It is essential for these same users to incorporate data stored on host systems into their regular correspondence, analysis, and reports.

Available methods of accessing host data do not provide the granular, "record-level" access required for cost-effective, secure, and meaningful integration of host and personal computer systems. In many cases, end users employ outright antiquated means of data integration. These methods include copying and pasting data from a terminal emulation screen, retyping information taken off printouts from host application reports, and importing text files containing comma-delimited values that use host EBCDIC-to-computer ASCII file transfer. These methods are not efficient although widely used and are not supported by products from independent software vendors (ISVs).

The challenge faced by IS professionals is how to provide direct record-level access to this valuable data without going through the host application. Much of the renewed interest in improved access to host data sources is a result of the burgeoning growth of local intranets, the use of the Internet, and Web technology as a mechanism for delivering information. Fast and inexpensive methods of record-level access are needed to deliver modern, three-tiered information systems during this era of cost-cutting and IS budget belt-tightening. Additional uses of this direct data access are ad hoc queries and Web-based reporting.

It is common for corporate management to rethink host data storage and the appropriate software used to provide data access. For many organizations, the answer to these issues is in rewriting the arguably outdated and certainly "misdated" host-based business rules with server-based, or even client-based, business logic.

The goal of the OLE DB Provider for AS/400 and VSAM is to provide customers and solution providers with the means to integrate desktop applications with this wealth of data residing on host computers.

The OLE DB Environment

Three main roles are performed by software applications in an OLE DB environment:

- **OLE DB Consumer**—The end-user or server-based program that uses (consumes) the OLE DB interfaces. An example is a Web-based component that makes OLE DB calls to integrate host records with a Web-based report.
- **OLE DB Data Provider**—A driver or other program that exposes OLE DB interfaces for use by consumer applications. Data providers translate OLE DB interfaces to a language or commands understood by the target data source. An example is the Microsoft® OLE DB Provider for AS/400 and VSAM, which translates OLE DB interfaces to DDM commands.
- **OLE DB Service Provider**—An application that both uses (consumes) and exposes OLE DB interfaces. Service providers typically act as proxies for the consumer, retrieving the data through the data provider and offering services to the consumer by manipulating the target data. An example is a query-processing engine.

DDM Record-Level Access

The Microsoft® OLE DB Provider for AS/400 and VSAM provides record-oriented access to host files. There is no need to perform bandwidth-intensive file transfers of entire host files to access data on the host.

The OLE DB interface provided by the OLE DB Provider for AS/400 and VSAM supports the following features:

- Set attributes and a record description of a host file (column information).
- Lock files and records.
- Position to the first record or the last record in a file.
- Navigate to the previous or next record in a file.
- Seek to a record, based on an index.
- Change records in a file.
- Insert new records and delete records in a file.
- Preserve file and record attributes.

The OLE DB Provider for AS/400 and VSAM is a "source" Distributed Data Management (DDM) requester implementation that can initiate DDM commands to be serviced by a remote host-based "target" DDM server. On the Microsoft Windows NT® operating system, the Microsoft DDM requester can run as a Windows NT service. This enables the DDM service to integrate with other host applications using the IBM DDM protocol and DDM servers resident on the host. Microsoft-based host software is not required (see Platforms Supported by the OLE DB Provider for AS/400 and VSAM). IBM offers DDM servers for the most popular host environments.

Providing users with direct record-level access reduces the development time to build and deploy new data integration solutions. Accessing only the target records, as opposed to entire host files, helps ensure data integrity.

Platforms Supported by the OLE DB Provider for AS/400 and VSAM

On the mainframe platform, IBM offers a target DDM server implementation in IBM Distributed File Manager (DFM), a component of IBM Data Facility Storage Management Subsystem (DFSMS). The Microsoft® OLE DB Provider for AS/400 and VSAM requires DFSMS version 1 release 2 or later for MVS/ESA and OS/390 to support an SNA LU6.2 connection.

On midrange AS/400 computers, IBM has implemented target DDM servers directly in OS/400. The Microsoft OLE DB Provider for AS/400 and VSAM requires OS/400 Version 3 Release 2 or later to support an SNA LU6.2 connection. The Microsoft OLE DB Provider for AS/400 and VSAM requires OS/400 Version 4 Release 2 or later to support a TCP/IP connection.

On the AS/400 platform, the OLE DB Provider for AS/400 and VSAM supports physical and logical files with an associated external record description file. For specific limitations, please see the *AS/400 DDM User's Guide*.

On the mainframe platform, the OLE DB Provider for AS/400 and VSAM supports the following data set types:

Sequential Access Method (SAM) data sets

- Basic Sequential Access Method data sets (BSAM)
- Queued Sequential Access Method data sets (QSAM)

Virtual Storage Access Method (VSAM) data sets

- Entry-Sequenced Data Sets (ESDS)
- Key-Sequenced Data Sets (KSDS)
- Fixed-Length Relative Record Data Sets (RRDS)
- Variable-Length Relative Record Data Sets (VRRDS)
- Relative Record Data Set (RRDS)
- VSAM Alternate Indexes for ESDS and KSDS data sets

Basic Partitioned Access Method (PDS) data sets

- Partitioned Data Set Extended members (PDSE)
- Partitioned Data Set members (PDS)
- Read-only support for PDSE directories
- Read-only support for PDS directories

The preceding data set types are supported by IBM DFM/MVS.

The following data set types are not supported by DFM/MVS and cannot be accessed using the OLE DB Provider for AS/400 and VSAM.

- VSAM Linear Data Sets (LDS)
- Generation Data Groups (GDG)
- Generation Data Sets (GDS)
- Basic Direct Access Method data sets (BDAM)
- Indexed Sequential Access Method data sets (ISAM)
- Sequential Data Striping data sets
- OpenEdition MVS Hierarchical File System (HFS) files
- Tape Media

All mainframe data sets accessible through IBM Distributed File Manager must be cataloged in an Intersystem communications function (ICF) catalog and reside on direct access storage devices (DASD).

The OLE DB Provider for AS/400 and VSAM supplied with Host Integration Server 2000 supports the following operating systems:

- Microsoft Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Professional

- Microsoft Windows NT® Server 4.0 with Service Pack 5 or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 5 or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 5 or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 5 or later
- Microsoft Windows 98, Second Edition

The OLE DB Provider for AS/400 and VSAM supplied with Host Integration Server 2000 Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

The OLE DB Provider for AS/400 and VSAM supplied with Host Integration Server 2000 supports only the Intel Windows 2000 and Windows NT platforms. Older versions of the OLE DB Provider for AS/400 and VSAM that shipped with SNA Server 4.0 Service Pack 1 and later supported Windows NT on the Alpha architecture.

The OLE DB Provider for AS/400 and VSAM requires the following computer-to-host connectivity software:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server End-User Client
- Microsoft Host Integration Server Administrator Client

Microsoft Host Integration Server 2000 can be installed on Windows 2000 Server, Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows NT 4.0 Server, Windows NT 4.0 Server Enterprise Edition, or Windows NT 4.0 Server Terminal Server Edition.

The Microsoft Host Integration Server Administrator Client can be installed on Windows 2000 Professional or Windows NT 4.0 Workstation. The Microsoft Host Integration Server Administrator Client with Service Pack 1 can also be installed on Windows XP Professional. The Administrator Client cannot be installed on Windows 98 or Windows 95.

The Microsoft Host Integration Server End-User Client can be installed on Windows 2000 Professional, Windows NT 4.0 Workstation, or Windows 98. The Microsoft Host Integration Server End-User Client with Service Pack 1 can also be installed on Windows XP Professional, Windows XP Home Edition, or Windows Millennium Edition.

The OLE DB Provider for AS/400 and VSAM supplied with Microsoft Host Integration Server 2000 supports the following OLE DB and ADO versions:

- OLE DB version 2.5. The Host Integration Server 2000 data access features require the runtime libraries for OLE DB version 2.5. These libraries must be installed prior to installing the OLE DB Provider for AS/400 and VSAM. On Windows 2000, these OLE DB libraries are installed as part of the Windows 2000 operating system. On Windows NT 4.0, Windows 98, and Windows 95, these library files must be installed by running the Microsoft Data Access Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.
  A version of OLE DB version 2.5 SDK is included in the Microsoft Data Access SDK which is available as a part of the Windows 2000 Platform SDK. These downloadable SDKs are available from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.
- ADO version 2.5. The Microsoft Host Integration Server 2000 data access features require the runtime libraries for ADO version 2.5. These libraries must be installed prior to installing the OLE DB Provider for AS/400 and VSAM. On Windows 2000, these ADO libraries are installed as part of the Windows 2000 operating system. On Windows NT 4.0, Windows 98, and Windows 95, these library files must be installed by running the Microsoft Data Access Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.
  A version of the ADO 2.5 SDK is included in the Microsoft Data Access SDK which is available as a part of the Windows 2000 Platform SDK. These downloadable SDKs are available from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.

OLE DB version 2.0 or later and ADO version 2.0 or later are required to support indexed record access from an ADO consumer application using the OLE DB Provider for AS/400 and VSAM. Indexed support through OLE DB is supported with OLE DB versions 2.0 and later.

Indexed File Access

The OLE DB Provider for AS/400 and VSAM provides both sequential and indexed file access. Sequential file access is provided for all supported file types on the Platforms Supported by the OLE DB Provider for AS/400 and VSAM.

Indexed file access is provided for the following host file types only:

- Mainframe Virtual Storage Access Method (VSAM) data sets.
    - Key-Sequenced Data Sets (KSDS) only when the keys are unique.
    - Fixed-length Relative Record Data Sets (RRDS) only when the keys are unique.
    - Variable-length Relative Record Data Sets (VRRDS) only when the keys are unique.
- AS/400 files.
    - Logical files.
    - Keyed physical files (externally described to the system).

OLE DB and ADO offer several interfaces that enable indexed file access.

The OLE DB Provider for AS/400 and VSAM supports integrated indexes based on the underlying rowset. OLE DB support for indexed file access using the OLE DB Provider for AS/400 and VSAM is available using the **IRowsetIndex**, **IViewFilter**, and **IViewRowset** interfaces.

For more information on indexes, see Chapter 8, "Indexes" and Chapter 16, "Integrated Indexes" in the *OLE DB Programmer's Reference*. To obtain a list of available indexes in a target AS/400 library, a program can call the OLE DB Session object's **IDBSchemaRowset::GetRowset** function requesting a query type of DBSCHEMA_INDEXES.

ADO support for indexed file access using the OLE DB Provider for AS/400 and VSAM is available using the Find method, Filter property, and Sort property on the ADO **Recordset** object. To obtain a list of available indexes in a target AS/400 library using ADO, a program can call the OpenSchema method on the **Connection** object specifying a *QueryType* of **adSchemaIndexes**.

By default, the OLE DB Provider for AS/400 and VSAM uses a server-based cursor. This means that all indexed file access is based on the cursor located over the host file, and not a local computer copy of the host file. If you want to use the many client-based cursor service providers available with the Microsoft® Data Access Components, then you must configure the provider to use a client-based cursor. For example, a client-based cursor is required when using Remote Data Service (RDS) and the Microsoft Visual Studio® ADO data-bound controls. However, using these controls, an application can access the host files for read-only purposes. If your application needs to access host files with an intent of both reading and writing and you require indexed file access, then your application should use the OLE DB Provider for AS/400 and VSAM's own server-based cursor.

File and Record Attributes

By definition, the record description is not part of the record I/O architecture in Distributed Data Management (DDM). Traditionally, applications must embed the record format as part of the application program. This creates a tremendous burden on the application and is very inconsistent with the existing computer-based data access standards, such as OLE DB and ODBC.

To solve this problem, the Microsoft® OLE DB Provider for AS/400 and VSAM uses an external Host Column Description (HCD) file stored on the computer that allows administrators to describe the host record format. At run time, the OLE DB Provider for AS/400 and VSAM transparently converts the host data to computer data using the local HCD information. Before a user program can view or open a VSAM file using the OLE DB Provider for AS/400 and VSAM, the user program must create a valid record description file or entry for the target VSAM file.

The OLE DB Provider for AS/400 and VSAM includes a Microsoft Management Console (MMC) application designed to enable administrators and developers to easily create these local record description files and the necessary registry settings for data sources. The OLE DB DDM Management application makes it relatively easy to create HCD files without ever knowing the HCD file format. The Host Column Description file format is documented in the Data Integration Reference.

The conversion process occurs in two steps. The host data is converted from host EBCDIC to ASCII data by the DDM DLL. The HCD file is used during this step to convert host data types to C data types, which are defined in ODBC and based on the SQL data types defined in the ANSI/ISO SQL-92 standard. The second phase of this conversion occurs in the SNAOLEDB DLL where these SQL C data types are converted to the defined OLE DB data types.

The use of an HCD file is not necessary to describe the record format for data stored in the AS/400 because the OLE DB Provider for AS/400 and VSAM automatically detects that the target host system is an AS/400 and uses the appropriate DDM commands to retrieve the record description. If the system administrator or the OLE DB application developer wants to use an HCD file instead of retrieving the AS/400 record description, this behavior can be forced by setting the configuration of the **Host Column Description File** property using Data Links. This parameter is described in the next section.

Configuring the OLE DB Provider for AS/400 and VSAM

Microsoft® Data Access Components 2.0 and later includes Data Links, a generic method for managing and loading connections to OLE DB data sources. Microsoft Data Links, a core element of the Microsoft Data Access Components (MDAC), provide a uniform method of creating persistent OLE DB data source object definitions stored in the form of universal data link (.udl) files. The OLE DB Provider for AS/400 and VSAM normally uses Data Links and UDL files for loading and configuring data sources.

Applications, such as the RowsetViewer sample from the Microsoft Data Access SDK, can open a UDL file that was previously created and pass the stored initialization string to the OLE DB Provider for AS/400 and VSAM at run time. Data Links provide a flexible method for finding and saving connection information to OLE DB data sources.

In order to use Microsoft OLE DB Provider for AS/400 and VSAM with an OLE DB consumer application, the user must either (1) create a Microsoft data link (UDL) file and call this from the application; or (2) call the OLE DB provider from within the application using a connection string that includes the provider name and other necessary parameters. If an application will be accessing VSAM data sets, then after configuring a data link, a host data description must also be configured using the Data Descriptions tool.

Microsoft Management Console (MMC) and MMC snap-ins are the current method of exposing administrative tasks and options in server-based Microsoft products. An MMC snap-in for the OLE DB Provider for AS/400 and VSAM is installed with the Host Integration Client 2000, which enables you to configure settings for accessing data files on AS/400 systems and mainframes. This OLE DB Management Console snap-in enables you to configure data descriptions used by the OLE DB Provider for AS/400 and VSAM.

The **OLE DB Provider for AS/400 and VSAM** console contains one high-level object:

- **Data Descriptions**—Stored in Host Column Description (HCD) files that contain the information required to convert host data types to computer data types.

The **OLE DB Provider for AS/400 and VSAM** console is designed to run on Microsoft Windows® 2000, Windows NT®, Microsoft Windows 98, and Microsoft Windows 95. On Windows 2000 and Windows NT, the console respects the Windows 2000 and Windows NT security hierarchy, where only privileged users can read and write to some areas of the system registry. To prevent general users from modifying the data sources and HCD files on Windows 2000 and Windows NT, the OLE DB Data Descriptions tool can only be run by users that have administrative privileges on the local computer.

Configuring Data Sources for the OLE DB Provider for AS/400 and VSAM

Data source information must be configured for each AS/400 or mainframe system data source object that is to be accessed using the OLE DB Provider. The default parameters for the OLE DB Provider are used as the default values for data sources and when these parameters are not configured for each data source.

Microsoft Data Links, a core element of the Microsoft Data Access Components, provides a uniform method for creating file-persistent OLE DB data source object definitions in the form of Universal Data Link (UDL) files. Applications, such as the RowsetViewer sample included with the Microsoft Data Access SDK and the MSDN Platform SDK, can open created UDL files and pass the stored initialization string to the OLE DB Provider for AS/400 and VSAM at run time.

Creating New Data Links for the OLE DB Provider for AS/400 and VSAM

UDL files are normally stored in a special folder located at:

C:\Programs Files\Common Files\System\Ole DB\data links

Microsoft Data Access Components 2.5 introduced a set of new OLE DB interfaces and functions to enumerate, create, and modify data link UDL files for configuring data sources. The NewSnaDS.exe utility provided as part of the OLE DB Provider for AS/400 and VSAM enables users to create and modify data links. This tool makes calls to the OLE DB Service Component Manager that provides these functions.

To create a new UDL file, run the NewSnaDS tool. This tool is installed in the System folder below the subdirectory where Microsoft Host Integration Server 2000 was installed. The default location where this tool is installed is the following:

C:\Program FilesHost Integration Server\System\NewSnaDS.exe

A shortcut for this tool is added to the **Programs** menu under the **Host Integration Server\Data Integration** folder with a name of OLE DB Data Sources. This shortcut is created when Host Integration Server 2000 or the Host Integration 2000 Client are first installed and support for data access is selected.

A shortcut entitled the **OLE DB Data Sources Browser** is also added to the **Programs** menu under the **Host Integration Server\Data Integration** folder. This shortcut opens Windows Explorer to the default directory where UDL files are stored:

C:\Programs Files\Common Files\System\Ole DB\data links

Using SNA Server 4.0 and older versions of the Microsoft Data Access Components (MDAC 2.1), it was possible to create a new UDL file by navigating to this folder using Windows Explorer. In the right pane of the Windows Explorer, a right-click would open a shortcut menu and enable you to create a **New Microsoft Data Link**.

In the past, a data link file could also be created using SNA Server 4.0 with a shortcut in the SNA Server 4.0 program folder. And the properties of a data link file could be edited by opening the file from Windows Explorer. The procedures used with SNA Server 4.0 to create a new UDL file have been deprecated and will not work with Microsoft Host Integration Server 2000, Windows 2000, and MDAC 2.5 or later.

Once a UDL file is created using the NewSnaDS tool, the file can be changed to a more appropriate name and copied to other client computers for use with the OLE DB Provider for AS/400 and VSAM.

A new data link file can be created with the NewSnaDS utility using the following procedure:

1. Click **Start**, point to **Programs**, and then point to **Host Integration Server**.
2. Point to **Data Integration**, and then click **OLE DB Data Sources** to run the NewSnaDS tool.

   A UDL file is created, and the **Data Link Properties** dialog box is displayed.
3. Select **Microsoft OLE DB Provider for AS/400 and VSAM** from the list of providers, and then configure the data source information as needed.
4. Click **OK** to save the data link.

By default, data links are created in the following folder:

C:\Program Files\Common Files\System\Ole DB\data links

However, a data link can be created in this location and moved to other client computers or folder, as needed.

Note that on Windows XP, Windows 2000, and Windows NT 4.0 using an NTFS partition, the file access permissions for this default folder are inherited from the System\Ole DB folder. The default file permissions allow full control by all members of the Users and Power Users groups in a Windows domain. Data link files may contain connection properties and configuration information that should be accessible only to specific users. For security reasons it is recommended that data link files be protected with an Access Control List (ACL) that restricts access to only appropriate users.

Windows 95, Windows 98, and Windows Millennium Edition do not include file systems that offer support for ACLs. Windows XP, Windows 2000, and Windows NT 4.0 can also be installed on FAT or FAT32 file systems lacking support for access control. In these cases, there is not protection available to protect any sensitive information stored in UDL files.

Browsing Data Sources for the OLE DB Provider for AS/400 and VSAM

By default, data links are created in the Program Files\Common Files\System\OLE DB\Data Links folder. A shortcut is provided in the Microsoft Host Integration Server 2000 program group.

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to **Data Integration**, and then click **OLE DB Data Source Browser**.

   Windows Explorer opens to the default location where UDL files are stored. The list of data links saved in the default location appears.

Configuring Data Links for the OLE DB Provider for AS/400 and VSAM

To edit the properties of a Data Link file, right-click the file using Windows Explorer and click **Data Link Properties**. The **Data Link Properties** dialog box appears with several property tabs:

- General
- Security
- Summary
- Provider
- Connection
- Advanced
- All

The **General**, **Security**, and **Summary** tabs provide access to general file information for the UDL file that is available for other files and is not related to the Data Link properties. This information includes file location, file type, file size, file dates, file security permissions for access, and descriptive summary information (description and origin properties and values such title, subject, author, etc.) for the UDL file. The **General** tab has a text box with the name of the Data Link. This filename must end with the .UDL extension if the file is to be recognized as a Data Link file. Note that the **Security** and **Summary** tabs are available on NTFS files systems, not on the older FAT file systems.

The `NewSnaDS` tool can also be used to open and modify an existing UDL file. The **Data Link Properties** dialog box appears with several property tabs:

Provider

Connection

Advanced

All

Provider

The **Provider** tab enables you to select the OLE DB Provider (the provider name string) to be used in this UDL file from a list of possible OLE DB providers. Select the Microsoft OLE DB Provider for AS/400 and VSAM. The parameters and fields displayed by the remaining tabs (**Connection**, **Advanced**, and **All**) are determined by the OLE DB Provider that is selected.

Connection

The **Connection** tab enables you to configure the basic properties required to connect to a data source. For the Microsoft OLE DB Provider for AS/400 and VSAM, the connection properties include the following values.

Property	Description
Data Source	The data source is an optional parameter that can be used to describe the data source. When the NewSnaDS configuration program is loaded from the Host Integration Server 2000 program folder, th e Data Source field is required. This field is used to name the UDL file, which is stored in the Program Files\Comm on Files\System\OLE DB\Data Links folder.
Location	The remote database name used for connecting to OS/400 systems. In DB2/400, this property is referred to as RD BNAM. This parameter is not used when connecting to mainframe systems.
Use Windows NT Integrate d security	This radio button enables using the Host Integration Security features providing a single-sign on to access this OL E DB data source. When this radio button is selected, the **User name** and **Password** fields are grayed out and become inaccessible. The user name and password fields are set based on the Windows 2000 logon values. When this radio button is not selected, the **User name** and **Password** fields must normally contain appropriate v alues to access data sources on hosts.
Use a specific user name an d password	This radio button disables using the Host Integration 2000 security features. When this radio button is selected, the user name and password fields are accessible and must normally contain a ppropriate values to access data sources on hosts.
User name	A valid user name and password are normally required to access data sources on hosts. These values are case sen sitive. The user must click the radio button that requires a specific user name and password to be entered.
Password	A valid user name and password are normally required to access data sources on hosts. These values are case sen sitive. The **Blank password** checkbox is only applicable for a Test Connection. In order to enter a password, the user will need to clear the **Blank password** check box if it is checked. If **Blank password** is checked, then a Test Connectio n with a blank password will not cause the OLE DB Provider to prompt for a password. Optionally, the user can choose to save the password in the UDL file by clicking the **Allow saving password** chec k box. Users and administrators should be warned that this option saves the authentication information (passwor d) in plain text within the UDL file.

It is possible to connect using a specific User name and Password defined on the host system or use the single sign-on feature (often referred to as Windows NT integrated security). If a specific User name and Password is to be used, this information may need to be saved into the UDL file. The User name and Password are saved in clear text in the UDL file. For security reasons in these cases, it is imperative that the UDL file be protected with an Access Control List (ACL) that restricts access to only authorized users. Saving the User name and Password in the data link also forces this UDL file to be updated whenever the Password associated with the User name is changed. So for a variety of reasons, specifying a User name and Password is not the preferred authentication option. Using the Windows NT integrated security option is the preferred method for authentication.

The **Connection** tab also includes a **Test Connection** button that can be used to test the connection parameters. The connection can only be tested after all of the required parameters are entered. When you click this button, an APPC session or a TCP/IP session attempts to be established to the host using the OLE DB Provider for AS/400 and VSAM.

Advanced

The **Advanced** tab enables you to set the network protection level and access permissions. You can set the protection level from the list box of allowable values. Access permissions are set by clicking the appropriate check boxes. For the Microsoft OLE DB Provider for AS/400 and VSAM, these properties include the following values:

Property	Description
Impersonation level	The level of impersonation that the server is allowed to use when impersonating the client. This property applies only to network connections other than Remote Procedure Call (RPC) connections. These impersonation levels are similar to those provided by RPC. The values of this property correspond directly to the levels of impersonation that can be specified for authenticated RPC connections, but can be applied to connections other than authenticated RPC. This parameter can be set to one of the following values: **Anonymous**—The client is anonymous to the server. The server process cannot obtain identification information about the client and cannot impersonate the client. **Delegate**—The process can impersonate the client's security context while acting on behalf of the client. The server process can also make outgoing calls to other servers while acting on behalf of the client. **Identity**—The server can obtain the client's identity. The server can impersonate the client for access control list (ACL) checking, but cannot access system objects as the client. **Impersonate**—The server process can impersonate the client's security context while acting on behalf of the client. This information is obtained when the connection is established, not for every call. This parameter defaults to **Impersonate**. The impersonation level parameter can also be set using the **All** tab.
Protection level	The level of protection of data sent between client and server. The values of this property correspond directly to the levels of protection that can be specified for authenticated RPC connections. This parameter can be set to one of the following values: **Call**—Authenticates the source of the data at the beginning of each request from the client to the server. **Connect**—Authenticates only when the client establishes the connection with the server. **None**—Performs no authentication of data sent to the server. **Pkt**—Authenticates that all data received is from the client. **Pkt Integrity**—Authenticates that all data received is from the client and that it has not been changed in transit. **Pkt Privacy**—Authenticates that all data received is from the client, that it has not been changed in transit, and protects the privacy of the data by encrypting it. This parameter is not supported by the OLE DB Provider for AS/400 and VSAM and defaults to connect-level protection. The protection level parameter can also be set using the **All** tab.
Connection timeout	The amount of time (in seconds) to wait for initialization to complete. This parameter is not currently supported by the OLE DB Provider for AS/400 and VSAM and defaults to 0. This parameter is equivalent to the DBPROP_INIT_TIMEOUT OLE DB property ID.
Access permissions	Once a connection is established, this parameter represents a bit mask of the access permissions that will be applied to the data file. As implemented by the OLE DB Provider for AS/400 and VSAM, access permissions apply to host file locks and do not apply to record locks. The allowable values include the following: Read, ReadWrite, Share Deny None, Share Deny Read, Share Deny Write, Share Exclusive, and Write. The default value for the **Access permissions** parameter for the OLE DB Provider for AS/400 and VSAM is ReadWrite. This parameter is equivalent to the DBPROP_INIT_MODE OLE DB property ID. These access permissions can also be set using the **All** tab.

All

The **All** tab allows users to configure essentially all of the properties for the data source except for the OLE DB Provider. The properties available in the **All** tab include properties that can be configured using the **Connection** and **Advanced** tabs as well as optional detailed properties used to connect to a data source. Some of the properties in the **All** tab are required.

These properties on the **All** tab may be edited by selecting a property from the list displayed and selecting **Edit Value**. This button will invoke dialog box for the specific property containing a Property Description describing the property and a Property Value box for making changes.

For the Microsoft OLE DB Provider for AS/400 and VSAM, these properties include the following values:

Property	Description
APPC Local LU Alias	The name of the local LU alias configured in the Host Integration Server 2000 computer. This parameter is equivalent to the DBPROP_SNAOLEDB_LOCALLU OLE DB property ID.
APPC Mode Name	When LU 6.2 (SNA) is selected for the Network Transport Library, this field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server 2000 computer configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP. This parameter is equivalent to the DBPROP_SNAOLEDB_APPCMODE OLE DB property ID.
APPC Remote LU Alias	When LU 6.2 (SNA) is selected for the Network Transport Library, this field is the name of the remote LU alias configured in the Host Integration Server 2000 computer. This parameter is equivalent to the DBPROP_SNAOLEDB_REMOTELU OLE DB property ID.
Cache Authentication	This parameter determines whether the OLE DB Provider caches authentication information such as a password in an internal cache. The value of this property (true or false) is selected from the drop-down list box. This parameter is not currently supported by the OLE DB Provider and defaults to false. This parameter is equivalent to the DBPROP_AUTH_CACHE_AUTHINFO OLE DB property ID.
Connect Timeout	The amount of time (in seconds) to wait for initialization to complete. This parameter is not currently supported by the OLE DB Provider and defaults to 0. This parameter is equivalent to the DBPROP_INIT_TIMEOUT OLE DB property ID.
Data Source	The data source is an optional parameter that can be used to describe the data source. This parameter is equivalent to the DBPROP_INIT_DATASOURCE OLE DB property ID.
Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files. This parameter is equivalent to the DBPROP_SNAOLEDB_LIBRARY OLE DB property ID.
Encrypt Password	This parameter determines whether special security mechanisms are used to ensure password privacy. The value of this property (true or false) is selected from the drop-down list box. This parameter is not currently supported by the OLE DB Provider and defaults to false. This parameter is equivalent to the DBPROP_AUTH_ENCRYPT_PASSWORD OLE DB property ID.

Extended Properties	This parameter is a string containing provider-specific, extended connection information. Properties passed through this parameter should be delimited by semicolons and will be interpreted by the provider's underlying network client. The use of this property implies that the OLE DB consumer knows how this string will be interpreted and used by the OLE DB provider. This parameter should be used only for provider-specific connection information that cannot be explicitly described through the other property parameters. This parameter is equivalent to the DBPROP_INIT_PROVIDERSTRING OLE DB property ID.
Host CCSID	The character code set identifier (CCSID) matching the data as represented on the host. The CCSID property is required when processing binary data as character data. Unless the **Process Binary as Character** value is set, character data is converted based on the host column CCSID and default ANSI code page. This parameter defaults to U.S./Canada (37). This parameter is equivalent to the DBPROP_SNAOLEDB_HOSTCCSID OLE DB property ID.
Host Column Description File	The fully qualified filename of the DDM Host Column Description (HCD) file. This parameter can be a UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the system directory below where the Host Integration Server or Client software was installed. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400. This parameter is equivalent to the DBPROP_SNAOLEDB_HCDPATH OLE DB property ID.
Impersonation Level	The level of impersonation that the server is allowed to use when impersonating the client. This property applies only to network connections other than Remote Procedure Call (RPC) connections; these impersonation levels are similar to those provided by RPC. The values of this property correspond directly to the levels of impersonation that can be specified for authenticated RPC connections, but can be applied to connections other than authenticated RPC. This parameter can be set to one of the following values: Anonymous—The client is anonymous to the server. The server process cannot obtain identification information about the client and cannot impersonate the client. Delegate—The process can impersonate the client's security context while acting on behalf of the client. The server process can also make outgoing calls to other servers while acting on behalf of the client. Identity—The server can obtain the client's identity. The server can impersonate the client for access control list (ACL) checking, but cannot access system objects as the client. Impersonate—The server process can impersonate the client's security context while acting on behalf of the client. This information is obtained when the connection is established, not on every call. The value of this property is selected from the drop-down list box. This parameter defaults to Impersonate. This parameter is equivalent to the DBPROP_INIT_IMPERSONATION_LEVEL OLE DB property ID.
Integrated Security	This parameter is a string containing the name of the authentication service used by the server to identify the user using the identity provided by an authentication domain. For example, for Microsoft® Windows 2000 Integrated Security, this is "SSPI" (for Security Support Provider Interface). If this parameter is a null pointer, the default authentication service should be used. When this property is used, no other DBPROP_AUTH* properties are needed and, if provided, their values are ignored. This parameter is equivalent to the DBPROP_AUTH_INTEGRATED OLE DB property ID.
Locale Identifier	This parameter specifies the locale to be used. This parameter is not supported by the OLE DB Provider for AS/400 and VSAM and defaults to 437. This parameter is equivalent to the DBPROP_INIT_LCID OLE DB property ID.
Location	The remote database name used for connecting to OS/400 systems. In DB2/400, this property is referred to as RDBNAM. This parameter is not used when connecting to mainframe systems. This parameter is equivalent to the DBPROP_INIT_LOCATION OLE DB property ID.

Mask Password	This parameter indicates whether the password should be sent to the data source or enumerator in a masked form. The value of this property (true or false) is selected from the drop-down list box. This parameter is not supported by the OLE DB Provider and defaults to false. This parameter is equivalent to the DBPROP_AUTH_MASK_PASSWORD OLE DB property ID.
Mode	Once a connection is established, this parameter represents a bit mask of the access permissions that will be applied to the data file. As implemented by the OLE DB Provider for AS/400 and VSAM, access permissions apply to host file locks and do not apply to record locks. The allowable values include the following: Read, ReadWrite, Share Deny None, Share Deny Read, Share Deny Write, Share Exclusive, and Write. This parameter can be a combination of zero or more of the following: DB_MODE_READ—Read-only. DB_MODE_WRITE—Write-only. DB_MODE_READWRITE—Read/write (DB_MODE_READ | DB_MODE_WRITE). DB_MODE_SHARE_DENY_READ—Prevents others from opening in read mode. DB_MODE_SHARE_DENY_WRITE—Prevents others from opening in write mode. DB_MODE_SHARE_EXCLUSIVE—Prevents others from opening in read/write mode (DB_MODE_SHARE_DENY_READ | DB_MODE_SHARE_DENY_WRITE). DB_MODE_SHARE_DENY_NONE—Neither read nor write access can be denied to others. This parameter is equivalent to the DBPROP_INIT_MODE OLE DB property ID.
Network Address	When TCP/IP has been selected for the Network Transport Library, this parameter is used to locate the target host computer. This parameter indicates the IP address or TCP/IP host name alias associated with the DDM server on the host. The network address is required when connecting through TCP/IP. This parameter is equivalent to the DBPROP_SNAOLEDB_NETADDRESS OLE DB property ID.
Network Port	When TCP/IP has been selected for the Network Transport Library, this parameter is used to locate the target DDM service access port when connecting through TCP/IP. This parameter represents the TCP/IP port used for communication with the DDM service on the host. The default value is TCP/IP port 446. This parameter is equivalent to the DBPROP_SNAOLEDB_NETPORT OLE DB property ID.
Network Transport Library	This parameter, which represents the dynamic-link library used for transport, designates whether the provider connects through SNA LU 6.2 or TCP/IP for network communication. The possible values for this parameter are TCPIP, or SNA. If TCPIP is selected, then values for Network Address and Network Port are required. TCP/IP connectivity to the mainframe is not supported by the OLE DB Provider for AS/400 and VSAM. If SNA is selected, then values for APPC Local LU Alias, APPC Mode Name, and APPC Remote LU Alias are required. The value of this property (SNA or TCPIP) is selected from the drop-down list box. This value defaults to SNA. This parameter is equivalent to the DBPROP_SNAOLEDB_NETTYPE OLE DB property ID.
Password	A valid user name and password are normally required to access data sources on hosts. The password is case sensitive and is shown as asterisks in this dialog box for security purposes. Optionally, you can choose to save the password in the UDL file by clicking the **Allow saving password** checkbox. Users and administrators should be warned that this option persists the authentication information in plain text within the UDL file. This parameter is equivalent to the DBPROP_AUTH_PASSWORD OLE DB property ID.

PC Code Page	Indicates the code page used for character code conversion. This property is required when processing binary data as character data. Unless the **Process Binary as Character** value is set, character data is converted based on the default ANSI code page configured in the Windows operating system. If this parameter is set to **Binary** or **65535**, then no character code conversions will take place. This parameter defaults to Latin 1 (1252). This parameter is equivalent to the DBPROP_SNAOLEDB_PCCODEPAGE OLE DB property ID.
Persist Security Info	This parameter indicates whether the data source object is allowed to persist sensitive authentication information such as a password along with other authentication information. Optionally, a user can choose to save the password in the UDL file by clicking the **Allow saving password** checkbox. Users and administrators should be warned that this option persists the authentication information in plain text within the UDL file. The value of this property (true or false) is selected from the drop-down list box. This parameter defaults to false. This parameter is equivalent to the DBPROP_AUTH_PERSIST_SENSITIVE_AUTHINFO OLE DB property ID.
Process Binary as Character	This parameter indicates whether to process binary fields (CCSID of 65535) as character data type fields on a per data source basis. The **Host CCSID** and **PC Code Page** values are required input parameters when this parameter is true. The value of this property (true or false) is selected from the drop-down list box. The default for this parameter is false, don't process binary fields as character fields. This parameter is equivalent to the DBPROP_SNAOLEDB_BINASCHAR OLE DB property ID.
Protection Level	The level of protection of data sent between client and server. The values of this property correspond directly to the levels of protection that can be specified for authenticated RPC connections. This parameter can be set to one of the following values: DB_PROT_LEVEL_NONE—Performs no authentication of data sent to the server. DB_PROT_LEVEL_CONNECT—Authenticates only when the client establishes the connection with the server. DB_PROT_LEVEL_CALL—Authenticates the source of the data at the beginning of each request from the client to the server. DB_PROT_LEVEL_PKT—Authenticates that all data received is from the client. DB_PROT_LEVEL_PKT_INTEGRITY—Authenticates all data received is from the client and that it has not been changed in transit. DB_PROT_LEVEL_PKT_PRIVACY—Authenticates all data received is from the client, that it has not been changed in transit, and protects the privacy of the data by encrypting it. The value of this property is selected from the drop-down list box. This parameter is not supported by the OLE DB Provider for AS/400 and VSAM and defaults to the connect level of protection. This parameter is equivalent to the DBPROP_INIT_PROTECTION_LEVEL OLE DB property ID.
Repair Host Keys	This parameter provides for repair of invalid key offsets received from OS/400 when keys have been defined using the DDS "RENAME" clause. This parameter indicates whether the OLE DB provider should repair any host key values set in the registry. This parameter defaults to false. The value of this property (true or false) is selected from the drop-down list box. This parameter is equivalent to the DBPROP_SNAOLEDB_REPAIRKEY OLE DB property ID.
Strict Validation	This parameter indicates whether strict validation should be used and defaults to false. The value of this property (true or false) is selected from the drop-down list box. This parameter is equivalent to the DBPROP_SNAOLEDB_STRICTVAL OLE DB property ID.

User ID	A valid user name is normally required to access data sources on hosts. This value is case sensitive. This parameter is equivalent to the DBPROP_AUTH_USERID OLE DB property ID.

Configuring Data Descriptions

The OLE DB Data Descriptions tool is a Microsoft Management Console (MMC) console that is used to describe the host data file format for mainframe access. Use of the **OLE DB for AS/400 and VSAM** console and the Host Column Description files is not required for AS/400 files because by default, the data file format is retrieved automatically from the host.

The data descriptions are stored in a local Host Column Description (HCD) file for each data source.

The following table provides the general parameters or attributes that can be configured describing each column in a data description on the **General** property page:

General parameters	Description
Name	The character string that represents the name of the column. This parameter may be null.
Alias	An optional character string that represents an alias label for the column string name. This parameter may be null.
Comment	An optional character string that represents a comment about the column. This attribute may be null.

The following table provides the data type parameters or attributes that can be configured describing each column in a data description in the **Data Type** property page:

Host parameters	Description
Type	The data type on the host. The allowed data values for host data type are selected from a drop-down list box (see Host Data Types for allowed values).
Length	Length of the field in bytes. Depending on the selected Host Type, this parameter may not be editable.
Precision	Total number of decimal digits in the column containing numeric data on the host. Depending on the selected host type, this parameter may not be editable. The only numeric data types that require this information are the PACKED and ZONED data types. And for these types, this field cannot be null; it must contain a valid numeric value. For all other host types, this parameter is not editable.
Scale	Number of decimal digits to the right of any decimal point for numeric data on the host. Depending on the selected host type, this parameter may not be editable. The only numeric data types that require this information are the PACKED and ZONED host data types. And for these types, this field cannot be null; it must contain a valid numeric value. For all other host types, this parameter is not editable.
CCSID	The character code set identifier (CCSID) used on the host. The allowed data values for host CCSID are selected from a drop-down list box (see allowed values listed below). This parameter defaults to the host CCSID configured for the OLE DB Provider and is typically U.S./Canada (37).

The following table provides the local parameter that can be configured:

Local parameters	Description
Type	Indicates the OLE DB data type on the local computer. The allowed data values for the local data type are selected from a drop-down list box (see Local OLE DB Data Types for allowed values).

The OLE DB provider limits the maximum length character field that can be accessed on an AS/400 computer to 32,745. On mainframes, a limitation of the IBM DFM is that SAM data sets and PDSE members are inaccessible if the fixed record length is greater than 32,760 or variable record lengths are greater than 32,756. DFM also limits all VSAM data sets on a mainframe to have a maximum record length no greater than 32,760. Attempting to access a character field greater than these lengths on an AS/400 or a mainframe machine will fail and can have unpredictable results.

The CCSID setting used by the OLE DB Provider must be set to match the CCSID actually used on the host—otherwise, data loss will occur. Some AS/400 systems default to a CCSID of 937 for enabling double-byte character sets (DBCS).

Host Data Types

The **Host Type** parameter represents the data type used for this column on the host. The allowed values for **Host Type** that can be selected from the drop-down list box include the following:

Host Type	Description
Binary	Fixed-length binary data (no character conversion). The length must be specified for this data type.
Character	Fixed-length character data. The length must be specified for this data type.
Date	The date represented as character data in the format yyyy-mm-dd that fits into 10 bytes.
DBCS	Fixed-length character data that can contain only DBCS data. The length must be specified for this data type.
DCBS – Mixed Either	Fixed-length character data that can contain either DBCS or alphanumeric data. The length must be specified for this data type.
DCBS – Mixed Open	Fixed-length character data that can contain both DBCS and alphanumeric data. DBCS data is distinguished from alphanumeric data with shift-control characters. The length must be specified for this data type.
DBCS – Variable	Variable-length character data with a prefix of 2 bytes for length that contains only DBCS data. The maximum possible length for the column containing this data type must be specified.
DCBS – Variable Mixed Either	Variable-length character data with a prefix of 2 bytes for length that can contain either DBCS or alphanumeric data. The maximum possible length for the column containing this data type must be specified.
DCBS – Variable Mixed Open	Variable-length character data with a prefix of 2 bytes for length that can contain both DBCS and alphanumeric data. DBCS data is distinguished from alphanumeric data with shift-control characters. The maximum possible length for the column containing this data type must be specified.
Double	Floating-point data that fits in 8 bytes (64 bits).
Long	Integer data that fits in 4 bytes (32 bits).
Packed	Packed decimal numeric data where the precision and scale are exactly as specified.
Short	Integer data that fits in 2 bytes (16 bits).
Single	Floating-point data that fits in 4 bytes (32 bits).
Time	The time represented as character data in the format hh:mm:ss that fits into 8 bytes.
Time Stamp	Time stamp represented as characters in the format yyyy-mm-dd hh:mm:ss.ffffff that fits into 19 bytes.
Variable Binary	Variable-length binary data represented as an unsigned character array with a prefix of 2 bytes for length. The maximum possible length for the column containing this data type must be specified.
Variable Character	Variable-length character data with a prefix of 2 bytes for length. The maximum possible length for the column containing this data type must be specified.
Zoned	Zoned decimal numeric data where the precision and scale are exactly as specified.

The floating-point data format assumed by the OLE DB Provider for AS/400 and VSAM depends on the host. For AS/400, the host floating-point data format is assumed to be IEEE. On mainframe hosts, floating-point data types are assumed to be in IBM floating-point formats. Because OLE DB supports the IEEE floating-point format, data conversion errors can occur when converting the extreme values of VSAM floating-point data in IBM format to IEEE floating-point data by the OLE DB Provider. These conversion errors occur because the default IBM floating-point formats and the IEEE floating-point format use a different number of bits for the mantissa and exponent when representing a floating-point number.

Local OLE DB Data Types

The **Local Type** represents the OLE DB data type used for this column on the computer. The OLE DB data types are defined in the OLE DB specifications and **#defines** can be found in the Oledb.h file. The allowed values for **Local Type** that can be selected from the drop-down list box include the following:

Host Type	Description
DBTYPE_BYTES	Fixed-length binary data represented as an unsigned char array.
DBTYPE_DBDATE	The OLE DB DBDATE typedef struct as defined in the Oledb.h file.
DBTYPE_DBTIME	The OLE DB DBTIME typedef as defined in the Oledb.h file.
DBTYPE_DBTIMESTAMP	The OLE DB DBTIMESTAMP typedef struct as defined in the Oledb.h file.
DBTYPE_DECIMAL	The OLE DB DECIMAL typedef struct as defined in the Oledb.h file.
DBTYPE_I2	Integer data stored in 2 bytes (16 bits).
DBTYPE_I4	Integer data stored in 4 bytes (32 bits).
DBTYPE_NUMERIC	The OLE DB NUMERIC typedef struct as defined in the Oledb.h file.
DBTYPE_R4	Single precision IEEE floating-point data stored in 4 bytes (32 bits).
DBTYPE_R8	Double precision floating-point data stored in 8 bytes (64 bits).
DBTYPE_STR	Fixed and variable length character data.

Converting Existing Data Sources

The OLE DB Management console that was previously used in SNA 4.0 and SNA 4.0 with Service Pack 1 for configuring OLE DB Provider for AS/400 and VSAM data sources has been removed and replaced by Microsoft Data Links and the Data Description Utility. Microsoft Data Links is a component of Microsoft Data Access Components (MDAC) 2.5. The MDAC 2.5 runtime must be installed prior to installing Host Integration Server 2000 when the OLE DB Provider for AS/400 and VSAM is selected to be installed. On Windows 2000, MDAC 2.5 is installed as part of the Windows 2000 operating system. On Windows NT 4.0 and Windows 98, these files must be installed by running the Microsoft Data Access Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.

Existing registry-based OLE DB Provider for AS/400 and VSAM data sources that were created in SNA Server 4.0 and SNA 4.0 SP1 can be converted to UDL files using the Reg2udl tool supplied with Host Integration Server. The Reg2Udl tool is not installed as part of Host Integration Server, but is located on the Host Integration Server 2000 CD-ROM in the \Options\Maintenance folder. Any UDL files that are converted should be moved to the following subdirectory:

C:\Program files\Common files\System\Ole db\data links folder.

When a duplicate UDL file is present in the destination folder, the Reg2udl tool will increment the file name by 1 (Data.udl will become Datat1.udl, for example). This may cause existing applications to fail because the OLE DB Provider for AS/400 and VSAM will be looking for the existing name (Data.udl).

Manual conversion of registry-based data sources to UDL files may be necessary in some cases when Setup for Host Integration Server 2000 is used (Web Setup for Windows 98 or Windows 95 is used, for example). A version of the Reg2udl tool (for Intel) can be found in the \Options\Maintenance folder on the Host Integration Server 2000 CD-ROM.

To convert data sources manually on SNA Server 4.0, use the appropriate Ireg2udl.exe (for Intel) or Areg2udl.exe (for Alpha) tool on the SNA Server 4.0 Service Pack 2 CD-ROM. On the SNA Server 4.0 Service Pack 3 CD-ROM, an Intel version of this tool for use on Windows NT 4.0, Windows, 98, and Windows 95 is located in the \Reg2Udl folder.

This Reg2Udl tool can also be used in a Microsoft Systems Management Server (SMS) package for rapid deployment of many client conversions at once.

Registry Settings Used by the OLE DB Provider for AS/400 and VSAM

The Microsoft® OLE DB Provider for AS/400 and VSAM uses a number of registry settings for configuration and proper operation. These registry settings are located under the **HKEY_LOCAL_MACHINE\Software\Microsoft\SNA OLE DB** key. These registry settings include the following subkeys:

Subkey Name	Description
Client	Stores the path to the directory where the Host Column Description (HCD) files are stored on the computer by the **OLE DB for AS/400 or VSAM** console.
Locale	Lists the conversions supported by the computer and the hosts.

The registry keys located under the client subkey include the following:

Value Name	Description
Path	Stores the path to the directory where Host Column Description (HCD) files are stored by the **OLE DB for AS/400 or VSAM**. This value defaults to the System directory below where Host Integration Server 2000 or SNA Server 4.0 was installed. On computers using Host Integration Server 2000 this value defaults to C:\Program Files\Host Integration Server\System. On computers using SNA Server 4.0, this value defaults to C:\SNA95\System on Microsoft Windows® 95 and Windows 98 and C:\SNA\System on Microsoft Windows NT®.

Registry subkeys located under the locale subkey indicate the character code conversions supported by the computer and the hosts and include the following:

Subkey	Descriptions
Host	Listed under this subkey are the possible CCSIDs for hosts when using custom code page conversion.
HostNLS	Listed under this subkey are the CCSIDs supported for hosts using SNA NLS.
PC	Listed under this subkey are the possible code pages when using custom code page conversions.

Programming Considerations Using the OLE DB Provider for AS/400 and VSAM

All the Microsoft® OLE DB objects exposed by the OLE DB Provider for AS/400 and VSAM support aggregation. Each OLE DB object has two classes, one that delegates its **IUnknown** calls and one that controls the object as a whole.

The apartment-threading model is supported, allowing multiple threads to access the objects safely. This is the only threading model supported.

When working with the Data Environment (DE) commands within Microsoft Visual Studio® 6.0, you must use a period (.) as a delimiter when specifying the AS/400 Library/File path. For example, the following is valid syntax when opening the AUTHORS physical file in the PUBS library on an AS/400:

```
EXEC OPEN PUBS.AUTHORS
```

In order to use the ADO Recordset Find method or the ADO Filter property, an AS/400 logical file, an AS/400 keyed physical file, a mainframe KSDS file with a unique key, or a mainframe RRDS file with a unique key must be used. If this method is used on an AS/400 non-keyed physical file or any other mainframe file type, then this method fails.

When using RRDS files, the **Find** method fails when a search is executed using a column name. For example, the following Visual Basic code will fail on an RRDS file with a column called Area:

```
RecordSet.Find "Area > '1111'", 0, adSearchForward, adBookmarkFirst
```

The error description will indicate that a Bookmark is invalid.

RRDS files don't not have an index based on a column name and the value of the column data, so the syntax to the above ADO **Find** method call doesn't make sense for RRDS files. In a COBOL program designed to dynamically find a record in an RRDS file, the record position would be passed. So for the 75th record in the file, a COBOL program would pass a value of 75. The COBOL program would then use the returned record number and the record length to calculate the position of the first byte of the record in the file.

The Data Environment's SQL command parser does not accept the forward slash (/) character. The OLE DB Provider for AS/400 and VSAM automatically substitutes a forward slash in place of the period and passes the correctly formatted path to your AS/400.

When using the Data Environment with Microsoft Visual Basic® 6.0, it is possible to get the following error when accessing the OLE DB Provider for AS/400 and VSAM:

```
"File is in use by another process. Unspecified error"
```

This error can occur once a data source has been configured for the OLE DB Provider for AS/400 and VSAM using the Data Links property page and a command is added using the Data Environment where the command added is the following:

```
"EXEC OPEN filename"
```

Using the Data Environment and selecting Run for this command can result in the above error. The Data Environment is opening the file and then trying to opening it a second time based on the command to execute without closing the first copy. Depending upon the share options of the dataset and the DBPROP_INIT_MODE property set for this data source, this error can occur and the user can be locked out of the AS/400 or VSAM file.

The OLE DB Provider for AS/400 and VSAM does not support the following SQL Server features:

- Data Transformation Services (DTS),
- Replication
- Distributed queries as a linked server.

When operating on large VSAM files and only querying data on a subset of the records, using the **Filter** property is not desirable because of the performance impact. The entire VSAM file is transferred to the client for filtering. A better solution is to use the server cursor engine and the **Find** method.

The syntax supported by the OLE DB Provider for AS/400 and VSAM for command text is as follows:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

The syntax supported by the OLE DB Provider for AS/400 and VSAM to open a rowset (table) using command text is as follows:

```
EXEC OPEN FileName
```

where *FileName* represents one of the following host file naming conventions:

Host file type	**File naming convention**
VSAM Data Sets	DATASETNAME.FILENAME
Partitioned Data Sets	DATASETNAME.FILENAME(MEMBER)
OS/400 Files	LIBRARY/FILE
OS/400 Files	LIBRARY/FILENAME
OS/400 File Members	LIBRARY/FILE(MEMBER)
OS/400 File Members	LIBRARY.FILENAME(MEMBER)

Note that if a member of a library contains a dot in the member name, the member name must be surrounded by double quotes. For example, if the member name is NAMES.DAT, the proper syntax used to open a rowset using command text is as follows:

```
EXEC OPEN LIBRARY/FILE("NAMES.DAT")
```

The distributed queries feature of SQL Server is sometimes referred to as the Distributed Query Processor (DQP).

Record Access and Data Conversion

The design of the OLE DB APIs is similar to the APIs provided by ODBC and other ISAM APIs. The APIs are handle-based. After opening a file, the application can determine the buffer size required to store a row, use the cursor APIs to move, and optionally retrieve one or more rows of data using the row-level binding.

Data is converted to default C data types as defined in ODBC, illustrated in the following table:

Host data type	**Default C data type**
Binary	unsigned char binary[]
Character	char string[]
Date (in character format)	date struct
Double	double
Long	int
Packed	unsigned char number[]
Short	short
Single	float
Time (in character format)	time struct
Time Stamp (in character format)	timestamp struct
Variable Binary	unsigned char binary[]
Variable Character	char string[]
Zoned	unsigned char number[]

Data conversions from a large numeric type to a small numeric type are supported (from DOUBLE to SINGLE and from INT to SMALLINT, for example), however truncation and conversion errors can occur that will not be reported by the OLE DB Provider for AS/400 and VSAM.

The OLE DB Provider for AS/400 and VSAM has a number of other limitations:

- Positive signed floating-point values cannot be read from ZONED DECIMAL fields.
- No floating point values can be inserted into ZONED DECIMAL fields.
- No values can be inserted into single-precision FLOATING POINT fields.
- Positive signed floating-point values cannot be inserted into PACKED DECIMAL fields.
- The ADO Find method fails to locate the first record when the key is multiple columns and the first column is a VARCHAR or TIME data type.

Record Locking

DDM supports record locks so that a requester can perform intended operations on a record without interference from concurrent users. Record locks are used only when the requester opens a file with an intent to update the file and specifies that the file is to be shared with another user. Two types of record locks are supported. Record locks are handled automatically by the Microsoft OLE DB Provider for AS/400 and VSAM whenever users call **IRowsetChange::SetData** (in immediate mode) or **IRowsetUpdate** (in the delayed mode). The OLE DB Provider for AS/400 and VSAM locks the record, updates the record, and then releases the lock.

Client Cursor Engines Using the OLE DB Provider for AS/400 and VSAM

The Microsoft Data Access Components (MDAC) supports the option of a client cursor engine. This feature is implemented as part of OLE DB, ADO, and Remote Data Services (RDS). When using ADO, a client cursor is enabled by setting the **CursorLocation** property on the recordset to **adUseClient**.

The OLE DB Provider for AS/400 and VSAM does not support any updating capabilities when used with a client cursor engine. In other words, if a client cursor engine is enabled using RDS or ADO, the OLE DB Provider cannot be used to update data on the host. The ADO recordset is treated as if it were read-only.

Single Sign-On

An Integrated Security (single sign-on) feature is supported by Microsoft Host Integration Server 2000 to automate the overall logon process. When configured for this feature, Host Integration Server 2000 automatically replaces special keywords in the data stream with the actual host user name and password at appropriate points in the session. This feature must be enabled by the administrator within a Host Integration Server 2000 subdomain and special strings must be entered for the user name (MS$SAME) and password (MS$SAME) that will be replaced.

When using this single sign-on feature with the OLE DB Provider for AS/400 and VSAM, the account in which the SNA DDM Service is running is used as the sign-on account. Thus, if the SNA DDM Service is running in the system account, single sign-on will always fail because the user name and password of the system account will be used rather than the actual user's name and password.

This feature can be exploited under certain circumstances when using the OLE DB Provider in combination with active server pages or other Web access schemes. If you want to use a single user account (UID) and password without revealing it through the Web page, the single sign-on provisions can be used so that the system service account UID and password are used.

Error Codes Returned by the OLE DB Provider for AS/400 and VSAM

The Microsoft OLE DB Provider for AS/400 and VSAM supports the following ranges of error codes:

Error code range	Source	Definition
1–100	Ddmapi.dll	OLE DB error codes (see the OLE DB Provider for AS/400 and VSAM help file).
256–511	Ddm.dll	IBM DDM documentation.
512–higher	Ddmwappc.dll	Errors specific to the OLE DB Provider for AS/400 and VSAM.

When using Host Integration Server 2000, passing an incorrect password at connect time yields "General Error" instead of "Authentication Error" (User does not have authority to access the host resource) on DB2/400 V4R4. An invalid local LU alias at connect time yields "Network Error" instead of "Invalid Local LU Alias" error. No error is reported when connecting using a non-existent default library value.

Using the OLE DB Provider for DB2

The Microsoft® OLE DB Provider for DB2 allows users to access IBM Data Base 2 (DB2) from within an OLE-aware application. The object linking and embedding database (OLE DB) is a standard set of interfaces that provides heterogeneous access to disparate sources of information located anywhere—file systems, e-mail folders, and databases. The OLE DB Provider for DB2 combines the universal data access of OLE DB with the IBM Distributed Relational Database Architecture (DRDA).

Organizations have invested in secure, robust, enterprise-wide data storage and management systems. DRDA is a set of rules for distributing or extending relational data from one computer to another, such as from a PC server to an IBM DB2 database server running on a mainframe or an AS/400 computer. By combining the OLE DB and DRDA architectures, Microsoft allows organizations to preserve their investments in existing data management infrastructure, while extending universal data access to all enterprise-wide data sources.

This section contains:

Goals of the OLE DB Provider for DB2

Relational database management systems (RDBMS) are one of the major sources of mission-critical information in today's enterprise organizations. Relational database technology enables departments and individual users to save their information in centrally-managed database stores that can be easily maintained by the organization's information systems group. Ad-hoc query tools designed for accessing relational database systems have added greater flexibility and ease of access to this information.

These same enterprises rely on vast networks of personal computers to enable their users to achieve business goals. End users invariably rely on network e-mail; Microsoft® Windows® productivity applications, such as Microsoft Office; and personal database programs, such as Microsoft Access, to accomplish their daily tasks. It is essential for these same users to incorporate data stored in relational database systems into their regular correspondence, analysis, and reports.

The challenge faced by IS professionals is how to provide access to this valuable data without the effort involved in developing traditional database applications. Much of the renewed interest in improved access to data sources is a result of the burgeoning growth in the use of Internet and Web technology as mechanisms for delivering information. Fast and inexpensive methods of accessing data stored in RDBMS systems are needed to deliver modern, three-tiered information systems during this era of cost cutting and IS budget belt-tightening. Additional uses of this relational database access include ad hoc queries and Web-based reporting.

IBM DB2 is a popular RDBMS for a significant number of enterprise customers. Customers need a cost-effective and manageable means to integrate DB2 with Microsoft SQL Server™, Microsoft Internet Information Services (IIS), and Microsoft Office applications. The goal of the OLE DB Provider for DB2 is to provide customers and solution providers with the means to integrate desktop applications with this wealth of data residing on DB2 database systems.

Distributed Relational Database Architecture

Database technology has allowed departments and even individual users to save their information locally—as opposed to centrally-managed stores owned by the organization's information systems group. Along with local storage and database query tools comes greater flexibility and ease of access to information. Yet, as more databases became distributed, a need emerged for users to access data stored remotely. IBM devised the Distributed Relational Database Architecture (DRDA) to enable their customers to access remote, distributed database systems across hardware platforms.

DRDA supports most dialects of the Structured Query Language (SQL) for access to relational database management systems (RDBMS). SQL is an international standard that defines a standardized language for accessing database management systems (DBMS). DRDA implementations generally support SQL in two ways: static (embedded) SQL, where the SQL commands are embedded directly in the application program and prepared as an extra step in the process of compiling the application; and dynamic or interactive (callable) SQL, where the user passes SQL commands as function calls at run time. One popular IBM implementation of dynamic SQL is the Call Level Interface (CLI). With dynamic SQL or CLI, SQL preparation is not required.

Clients that comply with DRDA are referred to as Application Requesters (AR) because they request data from the DRDA server. Servers that comply with DRDA are referred to as application servers (AS). Typically, application servers are implemented as the backend driver link to the RDBMS. In some cases, products are implemented as both application requesters and application servers.

DRDA supports access to stored procedures on DB2. SQL applications can invoke stored procedures or user-written programs on DB2 using the SQL CALL statement.

The OLE DB Provider for DB2 is an Application Requester implementation that can initiate DRDA commands to be serviced by a remote *target* DRDA application server represented by IBM DB2. On the Microsoft® Windows NT® operating system, the Microsoft DRDA application requester can run as a Windows NT service. This enables the integration of the DRDA service with other host applications using the IBM DRDA protocols and DRDA servers resident on the host. Microsoft host software is not required (see Platforms Supported by the OLE DB Provider for DB2). IBM offers Data Base 2 servers for most popular environments.

Platforms Supported by the OLE DB Provider for DB2

IBM and other software vendors have implemented DRDA support into database systems, such as IBM DB2, and database tools on a wide range of operating systems. DRDA is an open, published, and widely-supported protocol, which requires no additional license for development. This makes DRDA appealing to independent software vendors (ISVs), solution providers, large corporate development groups, as well as their customers.

The Microsoft® OLE DB Provider for DB2 is implemented as an IBM Distributed Relational Database Architecture (DRDA) application requester, which means it connects to popular DRDA-compliant DB2 systems.

The Microsoft OLE DB Provider for DB2 can access the following DB2 systems through SNA LU6.2 using Microsoft Host Integration Server 2000:

- DB2 for MVS Version 4 Release 1 (V4R1) or later
- DB2 for OS/390 Version 5 Release 1 (V5R1) or later
- DB2 for OS/400 (DB2/400) Version 3 Release 2 (V3R2) or later

The Microsoft OLE DB Provider for DB2 can access the following DB2 systems directly using TCP/IP:

- DB2 for OS/390 Version 5 Release 1 (V5R1) or later
- DB2 for OS/400 (DB2/400) Version 4 Release 2 (V4R2) or later
- DB2 Universal Database for Windows NT Version 5 Release 2 (V5R2) or later
- DB2 Universal Database for AIX Version 5 Release 2 (V5R2) or later

Note that DB2 for OS/400 (DB2/400) Version 4 Release 3 (V4R3) requires that PTF SF99103 be applied. DB2 for OS/400 (DB2/400) Version 4 Release 4 (V4R4) requires that PTF SF99104 be applied.

IBM DB2 with DRDA support is available on a variety of other platforms. The OLE DB Provider for DB2 has not been tested with these other implementations.

IBM DB2 for MVS Support

DB2 for MVS/ESA implements DRDA support in a component called Distributed Database Facility (DDF), which is an integral part of DB2 for MVS/ESA. IBM suggests that DDF provides optimal online transaction processing (OLTP) performance because of the close integration of DDF with the DB2 database engine. DB2 for MVS V4R1 and later includes a version of DDF that implements advanced DRDA features, such as stored procedures and distributed unit of work, including two-phase commit.

IBM DB2 for OS/400 Support

DB2 for OS/400 is built into OS/400 with no additional installation required. DB2 for OS/400 is a DRDA Application Requester supporting stored procedures and full distributed unit of work (two phase commit).

The Microsoft OLE DB Provider for DB2 can access the following DB2/400 systems through SNA LU6.2 using Microsoft Host Integration Server 2000:

- DB2 for OS/400 (DB2/400) Version 3 Release 2 (V3R2) or later

The Microsoft OLE DB Provider for DB2 can access the following DB2/400 systems directly using TCP/IP:

- DB2 for OS/400 (DB2/400) Version 4 Release 2 (V4R2) or later

Note that DB2 for OS/400 (DB2/400) Version 4 Release 3 (V4R3) requires that PTF SF99103 be applied. DB2 for OS/400 (DB2/400) Version 4 Release 4 (V4R4) requires that PTF SF99104 be applied.

IBM DB2 Universal Database Support

IBM DB2 Universal Database (UDB) is available on a variety of platforms. These implementations are also referred to as DB2 Common Server. The OLE DB Provider for DB2 supports accessing DB2 Universal Database V5R2 or later on Windows 2000, Windows NT, and IBM AIX directly using TCP/IP.

DB2 Common Server is also available for OS/2 and other versions of UNIX (HP HP-UX, Sun Solaris, Siemens-Nixdorf, and Bull). All of these DB2 Common Server implementations support DRDA Application Server capabilities, such as stored procedures and distributed unit of work.

Host Integration Server 2000 includes support for some ISO code pages for purposes of ISO-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-ISO, and ISO-to-UNICODE-to-ISO conversions when using the OLE DB Provider for DB2. These ISO code pages can be used when accessing DB2 Universal Database on AIX. For more information, see
Code Page Support Using the OLE DB Provider for DB2.

OLE DB Provider for DB2 Requirements

The OLE DB Provider for DB2 supplied with Host Integration Server 2000 supports the following operating systems:

- Microsoft® Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Professional
- Microsoft Windows NT® Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 6a or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later
- Microsoft Windows 98, Second Edition

The OLE DB Provider for DB2 supplied with Host Integration Server 2000 Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

The OLE DB Provider for DB2 supplied with Host Integration Server 2000 supports only the Intel Windows 2000 and Windows NT platforms. Versions of the OLE DB Provider for DB2 that shipped with SNA Server 4.0 Service Pack 1 and later supported Windows NT on the Alpha architecture.

The OLE DB Provider for DB2 requires the following computer-to-host connectivity software when connecting over SNA using LU 6.2:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server End User Client
- Microsoft Host Integration Server Administrator Client

Microsoft Host Integration Server 2000 can be installed on Windows 2000 Server, Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows NT 4.0 Server, Windows NT 4.0 Server Enterprise Edition, or Windows NT 4.0 Server Terminal Server Edition.

The Microsoft Host Integration Server Administrator Client can be installed on Windows 2000 Professional or Windows NT 4.0 Workstation. The Microsoft Host Integration Server Administrator Client with Service Pack 1 can also be installed on Windows XP Professional. The Administrator Client cannot be installed on Windows 98 or Windows 95.

The Microsoft Host Integration Server End-User Client can be installed on Windows 2000 Professional, Windows NT 4.0 Workstation, or Windows 98. The Microsoft Host Integration Server End-User Client with Service Pack 1 can also be installed on Windows XP Professional, Windows XP Home Edition, or Windows Millennium Edition.

Note that the OLE Provider for DB2 does not require any special host connectivity software when connecting directly to a host system using TCP/IP.

The OLE DB Provider for DB2 supplied with Microsoft Host Integration Server 2000 supports the following OLE DB and ADO versions:

- OLE DB version 2.5. The Host Integration Server 2000 data access features require the runtime libraries for OLE DB version 2.5. These libraries must be installed prior to installing the OLE DB Provider for DB2. On Windows 2000, these OLE DB libraries are installed as part of the Windows 2000 operating system. On Windows NT 4.0, Windows 98, and Windows 95, these library files must be installed by running the Microsoft Data Access Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.
  A version of OLE DB version 2.5 SDK is included in the Microsoft Data Access SDK which is available as a part of the Windows 2000 Platform SDK. These downloadable SDKs are available from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749, ADO version 2.5. The Microsoft Host Integration Server 2000 data access features require the runtime libraries for ADO version 2.5. These libraries must be installed prior to installing the OLE DB Provider for DB2. On Windows 2000, these ADO libraries are installed as part of the Windows 2000 operating system. On Windows NT 4.0, Windows 98, and Windows 95, these library files must be installed by running the Microsoft Data Access

Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749. A version of the ADO 2.5 SDK is included in the Microsoft Data Access SDK which is available as a part of the Windows 2000 Platform SDK. These downloadable SDKs are available from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749. The OLE DB Provider for DB2 has been tested with MDAC 2.6 runtime as it is shipped with Microsoft SQL Server 2000.

Configuring the OLE DB Provider for DB2

Microsoft® Data Access Components 2.0 and later includes Data Links, a generic method for managing and loading connections to OLE DB data sources. Microsoft Data Links, a core element of the Microsoft Data Access Components (MDAC), provide a uniform method of creating persistent OLE DB data source object definitions stored in the form of universal data link (UDL) files. The OLE DB Provider for DB2 normally uses Data Links and UDL files for loading and configuring data sources.

Applications, such as the RowsetViewer sample from the Microsoft Data Access SDK, can open created UDL files and pass the stored initialization string to the OLE DB Provider for DB2 at run time. Data Links provide a flexible method for finding and saving connection information to OLE DB data sources.

In order to use Microsoft OLE DB Provider for DB2 with an OLE DB consumer application, the user must either (1) create a Microsoft data link (UDL) file and call this from the application, or (2) call the OLE DB provider from within the application using a connection string that includes the provider name and any other needed parameters.

Configuring Data Sources for the OLE DB Provider for DB2

Data source information must be configured for each DB2 system data source object that is to be accessed using the OLE DB Provider for DB2. The default parameters for the OLE DB Provider for DB2 are used as the default values for data sources and when these parameters are not configured for each data source.

The Microsoft Data Links, a core element of the Microsoft Data Access Components, provides a uniform method for creating file-persistent OLE DB data source object definitions in the form of Universal Data Link (UDL) files. Applications, such as the RowsetViewer sample included with the Microsoft Data Access and the MSDN Platform SDK, can open created UDL files and pass the stored initialization string to the OLE DB Provider for DB2 at run time.

Creating New Data Links for the OLE DB Provider for DB2

UDL files are normally stored in a special folder located at:

C:\Programs Files\Common Files\System\Ole DB\data links

Microsoft Data Access Components 2.5 introduced a set of new OLE DB interfaces and functions to enumerate, create, and modify data link UDL files for configuring data sources. The NewSnaDS.exe utility provided as part of the OLE DB Provider for DB2 enables users to create and modify data links. This tool makes calls to the OLE DB Service Component Manager that provides these functions.

To create a new UDL file, run the NewSnaDS tool. This tool is installed in the **system** folder below the subdirectory where Microsoft Host Integration Server 2000 is installed. The default location where this tool is installed is the following:

C:\Program Files\Host Integration Server\system\NewSnaDS.exe

A shortcut for this tool is added to the **Programs** menu under the **Host Integration Server\Data Integration** folder with a name of **OLE DB Data Sources**. This shortcut is created when the Microsoft Host Integration Server 2000 or the Host Integration Client 2000 are first installed and support for data access is selected.

A shortcut entitled the **OLE DB Data Source Browser** is also added to the **Programs** menu in the **Host Integration Server\Data Integration** folder. This shortcut opens Windows Explorer to the default directory where UDL files are stored:

C:\Programs Files\Common Files\System\Ole DB\data links

Using SNA Server 4.0 and older versions of the Microsoft Data Access Components (MDAC 2.1), it was possible to create a new UDL file by navigating to this folder using Windows Explorer. In the right pane of the Windows Explorer, a right-click would open a shortcut menu and enable you to create a **New Microsoft Data Link**.

In the past, a data link file could also be created using SNA Server 4.0 with a shortcut in the SNA Server 4.0 program folder. And the properties of a data link file could be edited by opening the file from Windows Explorer. The procedures used with SNA Server 4.0 to create a new UDL file have been deprecated and will not work with Microsoft Host Integration Server 2000, Windows 2000, and MDAC 2.5 or later.

Once a UDL file has been created using the NewSnaDS tool, the file can be changed to a more appropriate name and copied to other client computers for use with the OLE DB Provider for DB2.

A new data link file can be created with the NewSnaDS utility using the following procedure:

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to **Data Integration**, and then click **OLE DB Data Sources** to run the NewSnaDS tool.

   A UDL file is created, and the **Data Link Properties** dialog box is displayed.
3. Select **Microsoft OLE DB Provider for DB2** from the list of providers, and then configure the data source information as needed.
4. Click **OK** to save the data link.

By default, data links are created in the following folder:

C:\Program Files\Common Files\System\Ole DB\data links

However, a data link can be created in this location and moved to other client computers or folder, as needed.

Note that on Windows XP, Windows 2000, and Windows NT 4.0 using an NTFS partition, the file access permissions for this default folder are inherited from the System\Ole DB folder. The default file permissions allow full control by all members of the Users and Power Users groups in a Windows domain. Data link files may contain connection properties and configuration information that should be accessible only to specific users. For security reasons it is recommended that data link files be protected with an Access Control List (ACL) that restricts access to only appropriate users.

Windows 95, Windows 98, and Windows Millennium Edition do not include file systems that offer support for ACLs. Windows XP, Windows 2000, and Windows NT 4.0 can also be installed on FAT or FAT32 file systems lacking support for access control. In these cases, there is not protection available to protect any sensitive information stored in UDL files.

Browsing Data Sources for the OLE DB Provider for DB2

By default, data links are created in following folder:

C:\Program Files\Common Files\System\Ole DB\data links

A shortcut is provided in the Microsoft Host Integration Server 2000 program group.

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to the Data Integration, and then click OLE DB Data Source Browser.

   Windows Explorer opens in the default location where UDL files are stored. The list of data links saved in the default location appears.

Configuring Data Links for the OLE DB Provider for DB2

To edit the properties of a Data Link file, right-click the file using Windows Explorer and click **Properties**. The **Properties** dialog box appears with several property tabs:

- **General**
- **Security**
- **Summary**
- Provider
- Connection
- Advanced
- All

The **General**, **Security**, and **Summary** tabs provide access to general file information for the UDL file that is available for other files and is not related to the Data Link properties. This information includes file location, file type, file size, file dates, file security permissions for access, and descriptive summary information (description and origin properties and values such title, subject, author, etc.) for the UDL file. The **General** tab has a text box with the name of the Data Link. This filename must end with the .UDL extension if the file is to be recognized as a Data Link file. Note that the **Security** and **Summary** tabs are available on NTFS files systems, not on the older FAT file systems.

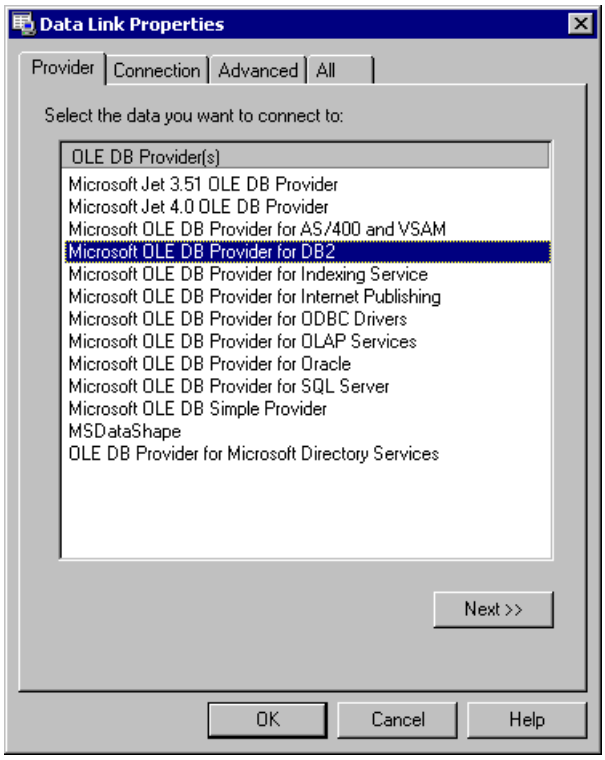
The **Provider**, **Connection**, **Advanced**, and **All** tabs provide access to the Data Link properties that need to be configured to connect to the DB2 system.

The `NewSnaDS` tool can also be used to open and modify an existing UDL file. In this tool, the **Data Link Properties** dialog box appears with the following property tabs:

- Provider
- Connection
- Advanced
- All

Provider

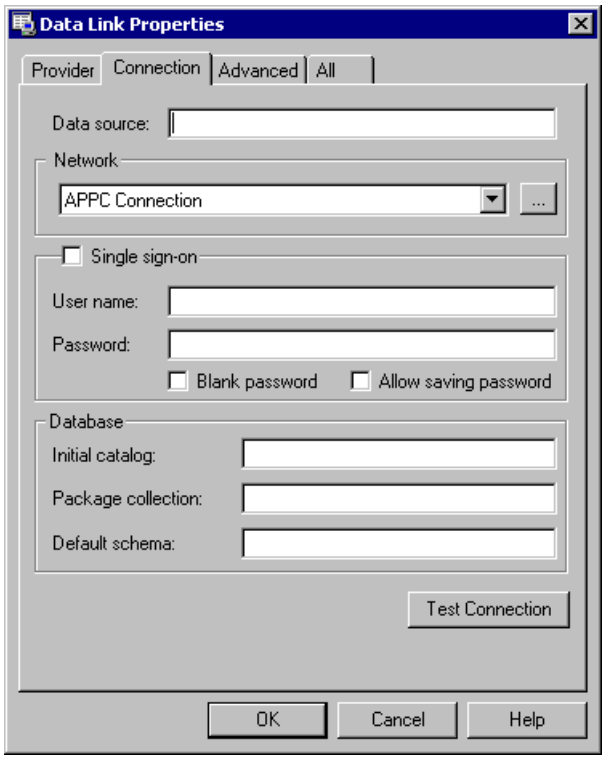
The **Provider** tab enables you to select the OLE DB provider (the provider name string) to use in the UDL file from a list of possible OLE DB providers. Select the Microsoft OLE DB Provider for DB2. The parameters and fields displayed in the remaining tabs (**Connection**, **Advanced**, and **All**) are determined by the OLE DB Provider that is selected.



Connection

The **Connection** tab enables you to configure the basic properties required to connect to a data source. The **Connection** tab dialog contains several sections:

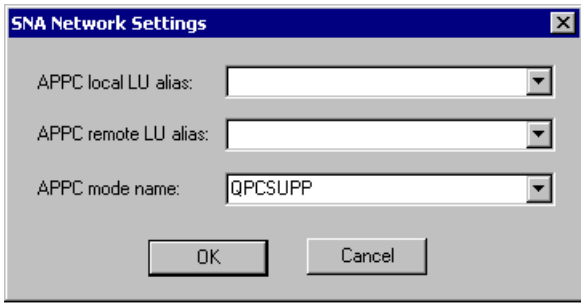
- Data source and Network connectivity
- Authentication
- Database Properties

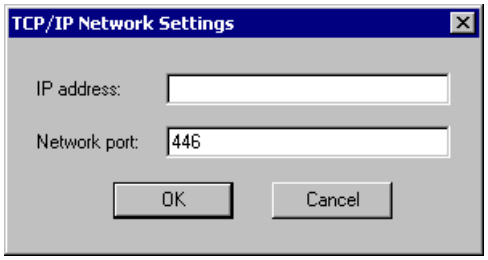


For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following properties for Data source and Network connectivity values:

Property	Description
Data source	The data source is an optional property that can be used to describe the data source. When the NewSnaDS configuration program is loaded from the Host Integration Server 2000 program folder, the Data source field is required. This field is used to name the UDL file, which is stored in the following default folder: C:\Program Files\Common Files\System\Ole DB\data links.
Network	This drop-down list box allows selecting the type of network connection to be used. The allowable options are **TCP/IP Connection** or **APPC Connection**. If **TCP/IP Connection** is selected, click the **More Options** (...) button, to open a dialog box for configuring TCP/IP network settings. The properties you can configure include the IP address of the DB2 host (or a hostname alias for this computer) and the Network Port (TCP/IP port) used for communication with the host. The default value for the Network Port is 446. The IP address of the host has no default value. If **APPC Connection** is selected (using SNA LU 6.2), click the **More Options** (...) button to open a dialog box for configuring APPC network settings. The properties you can configure include: the APPC local LU alias, the APPC remote LU alias, and the APPC mode name used for communication with the host. The default value for the APPC mode normally defaults to QPCSUPP. The local and remote LU alias fields do not have default values. The default value for the APPC mode name normally defaults to QPCSUPP. The APPC mode name can be selected from the drop-down list box.

The **Data source** in OLE DB is similar to a Data Source Name (DSN) in ODBC. The data source information is stored in a Microsoft Data Links file and contains the connection information required for the OLE DB Provider for DB2 to access IBM Data Base 2.





For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following properties for authentication information:

Property	Description
Single sign-on	Click this checkbox to enable using the Host Integration Security features providing a single-sign on to access this OLE DB data source. Note that single sign-on is only supported using the APPC Connection option (SNA LU 6.2). When this checkbox is selected, the User name and Password fields are grayed out and become inaccessible. The user name and password fields are set based on the login name used for the Windows 2000 or Windows NT 4.0 domain login. When this checkbox is not selected, the User name and Password fields must normally contain appropriate values in order to access data sources on hosts.
User name	A valid user name and password are normally required to access data sources on a host. These values are case-sensitive. Users must not check the Single sign-on option button if a specific user name and password are to be entered.
Password	A valid user name and password are normally required to access data sources on hosts. These values are case-sensitive. The **Blank password** checkbox is only applicable for a Test Connection. In order to enter a password, the user will need to clear the **Blank password** check box if it is checked. If **Blank password** is checked, then a Test Connection with a blank password will not cause the OLE DB Provider to prompt for a password. Optionally, users can choose to save the password in the UDL file by clicking the **Allow saving password** check box. Users and administrators should be warned that this option saves the authentication information (password) in plain text within the UDL file.

The AS/400 requires that the **User name** and **Password** properties be in uppercase. When connecting to DB2/400, these parameters must be passed as uppercase strings. When connecting to DB2 on IBM mainframes, the **User name** and **Password** parameters can be in mixed case.

It is possible to connect using a specific User name and Password defined in DB2 on the host system or use the Single sign-on feature (often referred to as integrated Windows security). If a specific DB2 user name and Password is to be used, this information may need to be saved into the UDL file. The User name and Password are saved in clear text in the UDL file. For security reasons in these cases, it is imperative that the UDL file be protected with an Access Control List (ACL) that restricts access to only authorized users. Saving the User name and Password in the data link also forces this UDL file to be updated whenever the Password associated with the User name is changed. So for a variety of reasons, specifying a User name and Password is not the preferred authentication option. Using the Single sign-on option is the preferred method for authentication.

For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following properties for database property values:

Property	Description

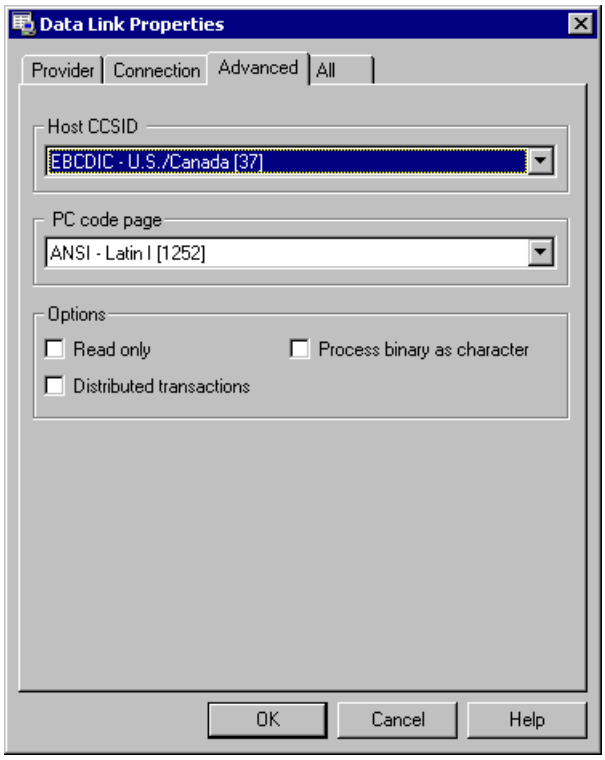
Initial catalog	This field is the first entry in the Database section of the Connection properties. This OLE DB property is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. If the provider supports changing the catalog for an initialized data source, the consumer can specify a different catalog name through the DBPROP_CURRENTCATALOG property in the DBPROPSET_DATASOURCE property set after initialization. This is a required property. This property is equivalent to the DBPROP_INIT_CATALOG OLE DB property ID.
Package collection	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. Package names are not restricted and can be upper case, lower case, or mixed case. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection property. By default, the OLE DB Provider will automatically create one package in the target collection, if one does not exist, at the time the user issues their first SQL statement. The package is created with GRANT EXECUTE authority to a single <AUTH_ID> only, where AUTH_ID is based on the User ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level specified when calling the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as when setting the ADO **IsolationLevel** property on the Connection object. A problem can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (the PUBLIC group on the DB2 system, for example), then the package is created for use only by this user. This means that other users may be unable to access the required package. The solution is for an administrative user with package administrative rights to create a set of packages for use by all users (see Creating Packages for Use with the OLE DB Provider for DB2). The OLE DB Provider for DB2 ships with a tool program for use by administrators to create packages. The crtpkg.exe tool is a Windows GUI application for use by the administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. This utility will create a set of packages and grant EXECUTE privilege on these packages to the PUBLIC group representing all users on the DB2 system . The packages (see descriptions under the isoLevel parameter of the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as the ADO IsolationLevel property) created are as follows: AUTOCOMMITTED package (MSNC001 is only applicable on DB2/400) READ UNCOMMITTED package (MSUR001) READ COMMITTED package, (MSCS001) REPEATABLE READ package, (MSRS001) SERIALIZABLE package (MSRR001) Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400. Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE, the DB2 for OS/400 QSYS2.SYSPACKAGE, and the DB2 Universal Database (UDB) SYSIBM.SYSPACKAGE catalog tables. Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000. The package names changed from SNA Server 4.0. This property is equivalent to the DBPROP_DB2OLEDB_PACKAGECOL OLE DB property ID.

Default schema	The name of the Collection where the OLE DB Provider for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The OLE DB Provider uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection. For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the OLE DB Provider uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined. This property is equivalent to the DBPROP_DB2OLEDB_CATALOGCOL OLE DB property ID.

The **Connection** tab also includes a **Test Connection** button that can be used to test the connection properties. The connection can only be tested after all of the required parameters are entered. When this button is pressed, an APPC session or a TCP/IP session will attempt to be established with the host using the OLE DB Provider for DB2.

Advanced

The **Advanced** tab allows users to select the character code set identifier used by the host, the PC code page used on the client, and select some specific options when using the OLE DB Provider for DB2.



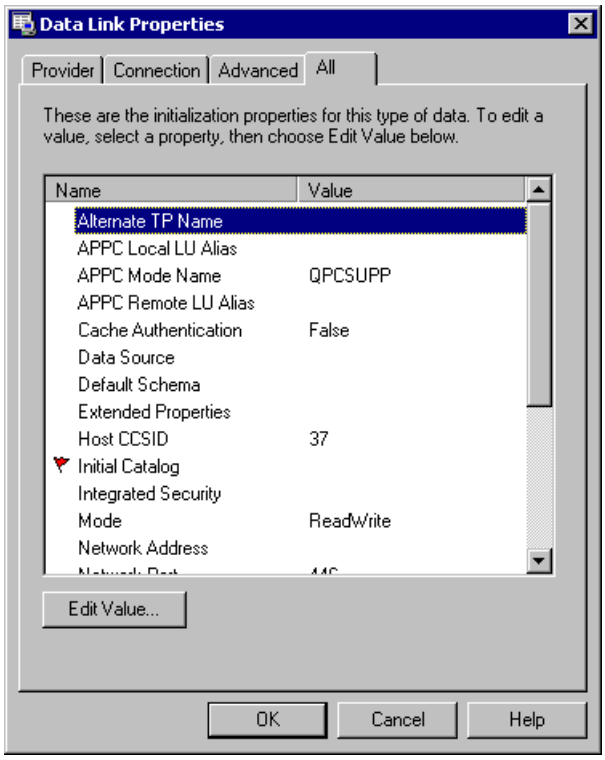
For the Microsoft OLE DB Provider for DB2, these properties include the following values:

Property	Description
Host CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote host computer. The CCSID property is required when processing binary data as character data. Unless the Process Binary as Character value is set to true, character data is converted based on the DB2 column CCSID and default ANSI code page. Note that Host CCSID 37 is not supported by the OLE DB Provider for DB2 when connecting to DB2 UDB for Windows NT or DB2 UDB for AIX. This property defaults to U.S./Canada (37). This property is equivalent to the DBPROP_DB2OLEDB_HOSTCCSID OLE DB property ID.
PC code page	The PC code page property indicates the code page to be used on the PC for character code conversion. This property is required when processing binary data as character data. Unless the Process binary as character checkbox is selected (value is set to true), character data is converted based on the default ANSI code page configured in Windows. This property defaults to Latin 1 (1252). This property is equivalent to the DBPROP_DB2OLEDB_PCCODEPAGE OLE DB property ID.
Read only	When this option is checked, the OLE DB Provider for DB2 creates a read-only data source by setting the Mode property to Read (DB_MODE_READ). A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This property defaults to a Mode property of Read/Write (DB_MODE_READ/WRITE). This property is equivalent to the DBPROP_INIT_MODE OLE DB property ID.
Process binary as character	When this option is checked (property is set to true), the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This property defaults to false. This property is equivalent to the DBPROP_DB2OLEDB_BINASCHAR OLE DB property ID.

Distributed transactions	When this option is checked, two-phase commit (distributed unit of work) is enabled. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. Th is option works only with DB2 for OS/390 V5R1 or later. This option also requires that an **APPC Connection** (the SNA LU 6.2 service) is selected as the network transport in the **Connection** tab and Microsoft Transaction Server (MTS) is installe d. This property is equivalent to the DBPROP_DB2OLEDB_UNITSOFWORK OLE DB property ID.

All

The **All** tab allows users to configure essentially all of the properties for the data source except for the OLE DB Provider. The properties available in the **All** tab include properties that can be configured using the **Connection** and **Advanced** tabs as well as optional detailed properties used to connect to a data source.



For the Microsoft OLE DB Provider for DB2, these properties include the following values:

Property	Description
Alternate TP Name	The Alternate Transaction Program (TP) Name property represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, the Alternative TP Name is set to 0X07F9F9F9. This property is equivalent to the DBPROP_DB2OLEDB_TPNAME OLE DB property ID.
APPC Local LU Alias	When an APPC Connection using SNA LU 6.2 is selected for the Network Transport Library, this field is the name of the local LU alias configured in the SNA server. This property is equivalent to the DBPROP_DB2OLEDB_LOCALLU OLE DB property ID.
APPC Mode Name	When an APPC Connection using SNA LU 6.2 is selected for the Network Transport Library, this field is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This property normally defaults to QPCSUPP. This property is equivalent to the DBPROP_DB2OLEDB_APPCMODE OLE DB property ID.

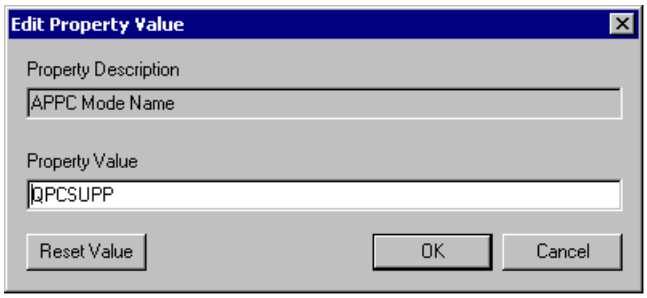
APPC Remote LU Alias	When an APPC Connection using SNA LU 6.2 is selected for the Network Transport Library, this field is the name of the remote LU alias configured in the SNA server. This property is equivalent to the DBPROP_DB2OLEDB_REMOTELU OLE DB property ID.
Cache Authentication	This property determines whether the OLE DB Provider caches authentication information. This property defaults to false. The value of this property (true or false) is selected from the drop-down list box. This property is equivalent to the DBPROP_AUTH_CACHE_AUTHINFO OLE DB property ID.
Data Source	The data source is an optional property that can be used to describe the data source. This property does not have a default value.
Default Schema	The name of the Collection where the OLE DB Provider for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The OLE DB Provider uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection. For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the OLE DB Provider uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined. This property is equivalent to the DBPROP_DB2OLEDB_CATALOGCOL OLE DB property ID.
Extended Properties	This property is a string containing provider-specific, extended connection information. The use of this property implies that the OLE DB consumer knows how this string will be interpreted and used by the OLE DB provider. This parameter should be used only for provider-specific connection information that cannot be explicitly described through the other property values. This property is equivalent to the DBPROP_INIT_PROVIDERSTRING OLE DB property ID.
Host CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote host computer. The CCSID property is required when processing binary data as character data. Unless the Process Binary as Character value is set to true, character data is converted based on the DB2 column CCSID and default ANSI code page. Note that Host CCSID 37 is not supported by the OLE DB Provider for DB2 when connecting to DB2 UDB for Windows NT or DB2 UDB for AIX. This property defaults to U.S./Canada (37). This property is equivalent to the DBPROP_DB2OLEDB_HOSTCCSID OLE DB property ID.

Initial Catalog	This OLE DB property is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. If the provider supports changing the catalog for an initialized data source, the consumer can specify a different catalog name through the DBPROP_CURRENTCATALOG property in the DBPROPSET_DATASOURCE property set after initialization. This is a required property. This property is equivalent to the DBPROP_INIT_CATALOG OLE DB property ID.
Integrated Security	This property determines whether the OLE DB Provider uses Host Security Integration (single sign-on). When this property is set to SSPI, single sign-on is enabled and separate user id and password parameters are not required. The user name and password fields are set based on the login name used for the Windows 2000 or Windows NT 4.0 domain login. When this property is null, this single sign-on feature is disabled. This property defaults to null (host security integration is disabled) and a user id and password are required. This property is equivalent to the DBPROP_AUTH_INTEGRATED OLE DB property ID.
Mode	A Mode property is a bit mask specifying access permissions. This bit mask can be a combination of zero or more of the following: DB_MODE_READ—Read-only. DB_MODE_READWRITE—Read/write (DB_MODE_READ | DB_MODE_WRITE). DB_MODE_SHARE_DENY_NONE—Neither read nor write access can be denied to others. DB_MODE_SHARE_DENY_READ—Prevents others from opening in read mode. DB_MODE_SHARE_DENY_WRITE—Prevents others from opening in write mode. DB_MODE_SHARE_EXCLUSIVE—Prevents others from opening in read/write mode (DB_MODE_SHARE_DENY_READ | DB_MODE_SHARE_DENY_WRITE). DB_MODE_WRITE—Write-only. The following values for mode are supported by the OLE DB Provider for DB2: Read (DB_MODE_READ) and Read/Write (DB_MODE_READ/WRITE). This property defaults to Read/Write. When the Read Only property is checked in the Advanced tab, the OLE DB Provider for DB2 creates a read-only data source by setting the Mode property to Read (DB_MODE_READ). A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This property is equivalent to the DBPROP_INIT_MODE OLE DB property ID.
Network Address	When TCP/IP has been selected for the Network Transport Library, this property indicates the IP address of the DB2 host or a hostname alias for this computer. This property is equivalent to the DBPROP_DB2OLEDB_NETADDRESS OLE DB property ID.
Network Port	When TCP/IP has been selected for the Network Transport Library, this property is the TCP/IP port used for communication with the DB2 host. The default value is TCP/IP port 446. This property is equivalent to the DBPROP_DB2OLEDB_NETPORT OLE DB property ID.

Network Transport Library	The network transport dynamic link library property designates whether the OLE DB Provider for DB2 connects via an APPC Connection using SNA LU6.2 or a TCP/IP Connection. The possible values for this property are TCPIP, or SNA. The default value for this property is SNA. If the default SNA is selected, then values for APPC Local LU Alias, APPC Mode Name, and APPC Remote LU Alias are required. If TCPIP is selected, then values for Network Address and Network Port are required. This property is equivalent to the DBPROP_DB2OLEDB_NETTYPE OLE DB property ID.
Package Collection	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. Package names are not restricted and can be upper case, lower case, or mixed case. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection property. By default, the OLE DB Provider will automatically create one package in the target collection, if one does not exist, at the time the user issues their first SQL statement. The package is created with GRANT EXECUTE authority to a single <AUTH_ID> only, where AUTH_ID is based on the User ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level specified when calling the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as when setting the ADO **IsolationLevel** property on the Connection object. A problem can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (the PUBLIC group on the DB2 system, for example), then the package is created for use only by this user. This means that other users may be unable to access the required package. The solution is for an administrative user with package administrative rights to create a set of packages for use by all users (see Creating Packages for Use with the OLE DB Provider for DB2). The OLE DB Provider for DB2 ships with a tool program for use by administrators to create packages. The crtpkg.exe tool is a Windows GUI application for use by the administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. This utility will create a set of packages and grant EXECUTE privilege on these packages to the PUBLIC group representing all users on the DB2 system . The packages (see descriptions under the isoLevel parameter of the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as the ADO IsolationLevel property) created are as follows: AUTOCOMMITTED package (MSNC001 is only applicable on DB2/400) READ UNCOMMITTED package (MSUR001) READ COMMITTED package, (MSCS001) REPEATABLE READ package, (MSRS001) SERIALIZABLE package (MSRR001) Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400. Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE, the DB2 for OS/400 QSYS2.SYSPACKAGE, and the DB2 Universal Database (UDB) SYSIBM.SYSPACKAGE catalog tables. Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000. The package names changed from SNA Server 4.0. This property is equivalent to the DBPROP_DB2OLEDB_PACKAGECOL OLE DB property ID.
Password	A valid user name and password are normally required to access data sources on hosts. The password is case-sensitive and is displayed as asterisks in this dialog box for security purposes. This property is equivalent to the DBPROP_AUTH_PASSWORD OLE DB property ID.
PC Code Page	The PC Code Page property indicates the code page to be used on the PC for character code conversion. This property is required when processing binary data as character data. Unless the Process Binary as Character value is set to true, character data is converted based on the default ANSI code page configured in Windows. This property defaults to Latin 1 (1252). This property is equivalent to the DBPROP_DB2OLEDB_PCCODEPAGE OLE DB property ID.

Persist Security Info	This property indicates whether the data source object is allowed to persist sensitive authentication information, such as a password along with other authentication information. This property defaults to false. The value of this property (true or false) is selected from the drop-down list box. This property is equivalent to the DBPROP_AUTH_PERSIST_SENSITIVE_AUTHINFO OLE DB property ID.
Process Binary as Character	When this property is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This property defaults to false. The value of this property (true or false) is selected from the drop-down list box. This property is equivalent to the DBPROP_DB2OLEDB_BINASCHAR OLE DB property ID.
Units of Work	This property indicates whether two-phase commit (distributed unit of work) used for transactions is supported for this data source. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. The following values for this property are supported by the OLE DB Provider for DB2: RUW (remote unit of work) DUW (distributed unit of work) This property defaults to RUW. Distributed unit of work (two-phase commit) works only with DB2 for OS/390 V5R1 or later. This option also requires that an APPC Connection using SNA LU 6.2 is selected as the network transport and Microsoft Transaction Server (MTS) is installed. This property is equivalent to the DBPROP_DB2OLEDB_UNITSOFWORK OLE DB property ID.
User ID	A valid User name is normally required to access data sources on hosts. This value is case-sensitive. This property is equivalent to the DBPROP_AUTH_USERID OLE DB property ID.

These properties on the **All** tab may be edited by selecting a property from the list displayed and selecting **Edit Value**. This button will invoke dialog box for the specific property containing a Property Description describing the property and a Property Value box for making changes.



Creating Packages for Use With the OLE DB Provider for DB2

The Microsoft® OLE DB Provider for DB2, which is implemented as an IBM Distributed Relational Database Architecture (DRDA) Application Requester, uses packages to issue SQL statements and call DB2 stored procedures. There is a provider-specific property that the OLE DB Provider for DB2 uses to identify a location in which to create and store DB2 packages. The OLE DB Provider for DB2 will create packages dynamically in the location to which the user points using the Package Collection property corresponding to the DBPROP_DB2OLEDB_PACKAGECOL property ID of OLE DB. This location may be configured using the Connection and Advanced tabs using Microsoft Data Links or can be passed as part of the connection string as an attribute keyword and argument. This attribute keyword can be either pkgcol or the long form of this attribute, Package Collection.

There are two package creation options:

1. The ODBC Provider for DB2 will auto-create one package for the currently-used isolation level at run-time if no package already exists. This auto-create process may fail if the user account does not have authority to create packages.
2. An administrator or user can manually creates all four packages (five packages on DB2/400) for use with all isolation levels and for use by all users (the PUBLIC group on DB2 representing all users) or a specific set of users. The OLE DB Provider for DB2 includes a utility program for use by users with appropriate administrative privilege that will create these packages and grant access to the PUBLIC group for this purpose.

However, some users may not have the security level when manually creating packages to GRANT authority to the packages to other users (grant authority to the DB2 PUBLIC group representing all users, for example). This can be a problem if two or more users with different user IDs try to access a single collection of packages. The first user that created the packages will have access to the packages, but the second user likely will not. The Microsoft® Host Integration Server 2000 CD-ROM includes a program for use by an administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. The Create Packages for DB2 utility, CrtPkg.exe, is a GUI-based tool included with Host Integration Server 2000 for creating packages for use with DB2. This tool is installed in the System folder below the subdirectory where the Microsoft Host Integration Server 2000 has been installed. The default location where this tool is installed is the following:

C:\Program Files\Host Integration Server\system\CrtPkg.exe

A shortcut for this tool is added to the **Programs** menu off the **Start** button on the Windows Taskbar under the **Host Integration Server\Data Integration** folder with a name of **Packages for DB2**. This shortcut is created when the Microsoft Host Integration Server or the Host Integration Client are first installed and support for Data Access is checked.

This tool will create a set of packages and grant EXECUTE privileges on these packages to the PUBLIC group. The PUBLIC group on DB2 systems is a default group that represents all DB2 users. The following packages are created:

- AUTOCOMMITTED package (MSNC001) is only applicable on DB2/400)
- READ UNCOMMITTED package (MSUR001)
- READ COMMITTED package (MSCS001)
- REPEATABLE READ package (MSRS001)
- SERIALIZABLE package (MSRR001)

Note that the AUTOCOMMITTED package (MSNNC001) is only created on DB2 for OS/400.

The descriptive process name used by the CrtPkg utility of each package corresponds with the isolation levels defined in the ANSI SQL standard. The table below indicates how these packages correspond with the terms used by IBM for isolation levels in DB2 documentation.

Package Description	**Package Name**	**IBM Documentation**
AUTOCOMMITTED (Note that this applies only to DB2/400 and does not correspond with an ANSI SQL isolation level	MSNC001	COMMIT(*NONE) (NC). This isolation level is used in DB2/400 auto-commit mode only and has no corresponding isolation level on other DB2 platforms or in ANSI SQL.
READ UNCOMMITTED	MSUR001	UNCOMMITTED READ (UR). This isolation level corresponds with ANSI SQL READ UNCOMMITTED.

READ COMMITTED	MSCS001	CURSOR STABILITY (CS). This isolation level corresponds with ANSI SQL READ COMMITTED.
REPEATABLE READ	MSRS001	READ STABILITY (RS). This isolation level corresponds with ANSI SQL REPEATABLE READ.
SERIALIZABLE	MSRR001	REPEATABLE READ (RR). This isolation level corresponds with ANSI SQL SERIALIZABLE.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000. The package names used by the OLE DB Provider for DB2 on SNA Server 4.0 are not compatible with the OLE DB Provider for DB2 included with Host Integration Server. On SNA Server 4.0, these packages used different names as follows:

```
AUTOCOMMITTED package (SNANC001) only applicable on DB2/400
READ UNCOMMITTED package (SNACH001)
READ COMMITTED package, (SNACS001)
REPEATABLE READ package, (SNARR001)
SERIALIZABLE package (SNAAL001)
```

These Isolation Levels are described in detail under Support for Isolations Level using the OLE DB Provider for DB2. These Isolation Levels are also described under the OLE DB isoLevel parameter and ADO IsolationLevel property. Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400.

Note that the CrtPkg tool creates this set of packages and grants EXECUTE privileges to the PUBLIC group. There may be cases for security reasons where EXECUTE privileges to this set of packages on the DB2 system should be restricted to a different group of users or specific users. In these cases, execution privileges on these created packages will need to be modified on the host system.

The CrtPkg utility will create all of these packages inside the Collection that is specified in the Package Collection property in the datalink file, or in the connection string. If the user does not have the appropriate authority to create packages in the specified Collection, or if the specified Collection does not exist, the OLE DB Provider for DB2 will return an error.

In the case of DB2 on MVS or OS/390, the normal error text returned if the user does not the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your specific DB2 version for
a description of the associated Native Error and SQL State. SQLSTATE: 51002, SQLCODE: -567.
```

In the case of DB2/400, the normal error text returned if the user does not the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your specific DB2 version for
a description of the associated Native Error and SQL State. SQLSTATE: 51002, SQLCODE: -805.
```

In the case of DB2/400, the normal error returned if the collection does not exist would be as follows:

```
Failed to create AUTOCOMMITTED (NC) package.  RETCODE=-99.
SQL Error: Code=-204, State=42704, Error Text= A SQL error has occurred.  Please consult the
documentation for your specific DB2 version for a description of the associated Native Error
and SQL State. SQLSTATE: 42704, SQLCODE: -204
```

There are two authorities required to execute the create package process on MVS using the CrtPkg utility:

```
GRANT BINDADD TO <authorization ID>
GRANT CREATE IN COLLECTION <collection ID> TO <authorization ID>
```

The "authorization ID" is the user who needs the permission to create the packages. The "collection ID" is the name of the Collection, which the user specifies in the datalink file for the Package Collection property. This Collection should be a valid Collection within the DB2.
If an administrator executes the above statements on behalf a non-privileged user, then this non-privileged user can then run the CrtPkg utility. Once run, the CrtPkg process will create four sets of packages (one for each of the four isolation levels supported

on DB2 for MVS or OS/390) for use by "all" (PUBLIC) users of the Microsoft data access features.

The example below illustrates this process on DB2 for MVS or DB2 for OS/390.

Grant rights to run the CrtPkg utility to authorization ID WNW999

```
GRANT BINDADD TO WNW999
GRANT CREATE IN COLLECTION MSPKG TO WNW999
```

Run the CrtPkg utility using authorization ID WNW999 (see output from CrtPkg below)

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

In order to execute the CrtPkg utility on DB2/400, a user ID must have one of the following authorities:

- *CHANGE authority on the DB2 collection
- *ALL authority on the DB2 collection

If the user merely has *USE authority or if the user has *EXCLUDE authority, the Create Package process will fail.

There are several steps required to change user authority on a DB2/400 collection (AS/400 library): From interactive SQL (STRSQL command) while logged in as user with administrative privileges, create a new collection. This command can also be issued using ADO, OLE DB, and ODBC. However, most administrators typically create collections from the AS/400 console since the administrator must be logged in at the console to issue the Command Language (CL) command with which to change the user authority on the collection.

```
CREATE COLLECTION <collection ID>
```

From the AS/400 command console, issue the CL WRKOBJ command with the <collection ID> as a parameter.

```
WRKOBJ <collection ID>
```

The "collection ID" is the name of the Collection, which the user specifies in the datalink file for the Package Collection property. This Collection should be a valid Collection within DB2. The Work with objects screen appears. Place the cursor on the *PUBLIC Object Authority line and change the authority from *USE to *ALL.

If an administrator executes the above statements on behalf a non-privileged user, then this non-privileged user can then run the CrtPkg utility. Once run, the CrtPkg process will create five sets of packages (one for each of the five isolation levels supported on DB2/400) for use by "all" (PUBLIC) users of the Microsoft data access features. On DB2/400, five packages are created including

the AUTOCOMMITTED packages.

The example below illustrates this process on DB2/400.

Grant rights to run the CrtPkg utility to authorization ID WNW999

```
CREATE COLLECTION MSPKG
WRKOBJ MSPKG
```

Run the CrtPkg utility (see the output from CrtPkg for DB2/400 below)

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating AUTOCOMMITTED (NC) package...
AUTOCOMMITTED (NC) package created.
Package creation succeeded.
EXECUTE privilege on MSNC001            granted to PUBLIC
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

CrtPkg allows a user to create a new UDL file or load a data source and modify an existing UDL file for connection configuration information. The File menu of CrtPkg has a New option used for creating a new OLE DB UDL File and a Load Data Source option to load an existing UDL file. The File menu Edit Data Source option allows a user to access and modify the properties for a data source similar to using the NewSnaDS.exe tool. The Run menu option is used to create packages.

When using the create package tool, if the package collection specified does not exist, then DB2 returns SQLCODE -805.

When using auto-create packages, if a package collection is not specified or the package collection does not exist, then during the "auto-create" package process, the consumer application will receive SQLSTATE HY000 and SQLCODE -385. The SQLSTATE HY000 is defined as a provider-specific error. The -385 Error Return Code is not a SQLCODE but rather a DDM DRDA AR (DB2 client) return code. This error code is defined as DDM_VALNSPRM with the following associated text string:

```
"The parameter value is not supported by the target system."
```

The OLE DB Provider for DB2 client error codes are defined in the db2oledb.h file located on the Host Integration Server 2000 CD-ROM.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000.

SNA Server 4.0 with Service Pack 3 came with two similar utilities for creating packages: CRTPKG.EXE (a command-line tool) and CRTPKGW.EXE (a GUI-based tool).

Registry Settings Used By the OLE DB Provider for DB2

The Microsoft® OLE DB Provider for DB2 uses a number of registry settings for configuration and proper operation. The configuration registry settings are located under the **HKEY_LOCAL_MACHINE\Software\Microsoft\SNA Server\CurrentVersion\Setup** key. These registry settings include the following subkeys:

Subkey	Comment
RootDir	Stores the path to the root directory where the Host Integration Server was installed. The system directory below this root directory is the location where the OLE DB Provider for DB2 DLLs and other support DLLs are installed.

Programming Considerations Using the OLE DB Provider for DB2

The Microsoft® OLE DB Provider for DB2 provides passthrough support for SQL statements. No SQL parsing is provided. The user must know what SQL syntax is supported for the target DB2 implementation. For information on what SQL syntax is supported, see the specific DB2 SQL Reference and DB2 Application Programming and SQL Guide for the DB2-specific platform.

The OLE DB Provider for DB2 does not parse the SQL statements to qualify table names. Consequently, users of the OLE DB Provider for DB2 must use either two-part or three-part (fully-qualified) object names when naming tables, views, and stored procedures in DB2. A two-part table name would consist of the user ID and table, <UserID>.<Table>. One-part names (just the table name) will not succeed unless the combination of the DB2 collection and schema name correspond directly to the OLE DB User ID (the OLE DB DBPROPSET_DBINIT property is equal to the OLE DB DBPROP_AUTH_USERID property).

All the OLE DB objects exposed by the OLE DB Provider for DB2 support aggregation. Each OLE DB object has two classes, one that delegates its **IUnknown** calls and one that controls the object as a whole.

The free-threading model is supported, allowing multiple threads to access the objects safely.

The current implementation of the OLE DB Provider for DB2 services all OLE DB Session, Command, and Rowset objects present in a given instance of the DataSource object through a single APPC conversation or TCP/IP connection. One implication of this design is that if two Rowset objects, each created from a different OLE DB Session object, use explicit commitment control through the **ITransaction** interface, they will interfere with each other. When a Commit or Abort for one instance is invoked, all work for the DataSource object will be either committed or aborted. This may yield undesirable results. The work around to this problem is to instantiate two instances of the DataSource object.

The OLE DB Provider for DB2 does not work with OLE DB Session Pooling.

The OLE DB Provider for DB2 in Microsoft Host Integration Server 2000 supports distributed transactions, DRDA Distributed Unit of Work, and can participate in a distributed transaction coordinated by Microsoft Distributed Transaction Coordinator. This feature is only available when connecting to one of the following across an LU 6.2 network connection:

- DB2 for OS/390 V5R1 or later
- DB2/400 V4R3 or later

This option also requires that the SNA LU 6.2 service is selected as the network transport and Microsoft Transaction Server (MTS) is installed. The Microsoft OLE DB Provider for DB2 does not support OLE DB automatic transaction enlistment under Microsoft Transaction Server. Note that the OLE DB Provider for DB2 supplied with SNA Server 4.0 does not support distributed transactions.

The Microsoft Data Access Components (MDAC) support the option of using a client cursor engine. This service component is implemented as part of OLE DB, ADO, and Remote Data Services (RDS). When using ADO, a client cursor is enabled by setting the CursorLocation property on the recordset to adUseClient. When using the ADO Client Cursor Engine with DB2 for OS/390, the developer must set the OLE DB Provider for DB2 Auto Commit Mode property in the data link or connection string to FALSE. This is not required when connecting to DB2 for OS/400.

The OLE DB Provider for DB2 included with Host Integration Server 2000 supports updating capabilities when used with a client cursor engine when the following requirements are met:

- To support updates (UPDATE, INSERT, and DELETE) using a client cursor engine, the values in at least one column in the target table must be unique.

When used with the version of the OLE DB Provider for DB2 provided with SNA Server 4.0 Service Pack 3 or later, the OLE DB Provider supports updating capabilities when used with a client cursor engine when the following requirements are met:

- To support updates (UPDATE, INSERT, and DELETE) using a client cursor engine, the values in at least one column in the target table must be unique.
- The Commit parameter must be set to FALSE (auto commit is off) when configuring the data source or when this parameter is passed as part of a connection string.

Previous versions of the OLE DB Provider for DB2 prior to SNA Server 4.0 Service Pack 3 do not support any updating capabilities when used with a client cursor engine. In other words, if a client cursor engine is enabled using RDS or ADO, the OLE DB Provider for DB2 cannot be used to update data on the host. The ADO recordset is treated as if it were read-only.

When the intent is to update records, DB2 requires that the SQL SELECT statement also include the FOR UPDATE option. For

example, to select all records from the AUTHORS table in the DB2 collection called PUBS with an intent to update requires the following SQL syntax:

```
SELECT * FROM PUBS.AUTHORS FOR UPDATE
```

When using DB2 for MVS V4R1 and DB2 for OS/400 V3R2, there are further requirements to indicate the columns that you intend to update. For example, to update the AU_LNAME and AU_FNAME columns in the PUBS.AUTHORS table, the following SQL syntax must be used:

```
SELECT * FROM PUBS.AUTHORS FOR UPDATE OF AU_LNAME, AU_FNAME
```

Microsoft Visual Studio® 6.0 offers a number of ADO data-bound controls, including a datagrid and the ADO Data Control. When using these ADO data controls, the developer must set the CursorLocation property on the recordset to adUseClient. Additionally, when using these ADO data controls with DB2 for OS/390, the developer must set the OLE DB Provider for DB2 Auto Commit Mode property in the data link or connection string to FALSE.

Support for Isolation Levels Using the OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2 provides flexibility in dealing with issues of isolation levels and transaction state. The isolation level for a session can be set using the DBPROP_SESS_AUTOCOMMITISOLEVELS property on a session.

The OLE DB Provider for DB2 supports the following values for the DBPROP_SESS_AUTOCOMMITISOLEVELS and DBPROP_SUPPORTEDTXNISOLEVELS property.

OLE DB Property Value	Description
DBPROPVAL_TI_BROWSE	This isolation level is the same as DBPROPVAL_TI_READUNCOMMITTED. Note that the OLE DB specification and the OLEDB.H include file defines two macros with the same value.
DBPROPVAL_TI_CHAOS	An undefined value for isolation level. This value is not supported using the OLE DB Provider for DB2.
DBPROPVAL_TI_CURSORSTABILITY	This isolation level is the same as DBPROPVAL_TI_READCOMMITTED. Note that the OLE DB specification and the OLEDB.H include file defines two macros with the same value.
DBPROPVAL_TI_ISOLATED	This isolation level is the same as DBPROPVAL_TI_SERIALIZABLE. Note that the OLE DB specification and the OLEDB.H include file defines two macros with the same value.
DBPROPVAL_TI_READCOMMITTED	When this property is set, it isolates any data read from changes by others and changes made by others by others cannot be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read. This is the default value for isolation level. This isolation level is also called Cursor Stability (CS) in IBM DB2 documentation. This isolation level corresponds with the ADO property set to adXactReadCommitted.
DBPROPVAL_TI_READUNCOMMITTED	When this property is set, it does not isolate data read from changes by others and changes made by others by others can be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read. This isolation level is called Uncommitted Read (UR) in IBM DB2 documentation. This isolation level corresponds with the ADO property set to adXactReadUncommitted.
DBPROPVAL_TI_REPEATABLEREAD	When this property is set, it isolates any data read from changes by others and changes made by others cannot be seen. The re-execution of the read statement is affected by others. This supports a repeatable read. This isolation level is called Read Stability (RS) in IBM DB2 documentation. This isolation level corresponds with the ADO property set to adXactRepeatableRead.
DBPROPVAL_TI_SERIALIZABLE	When this property is set, it isolates any data read from changes by others and changes made by others by others cannot be seen. The re-execution of the read statement is not affected by others. This supports a repeatable read. This isolation level is called Repeatable Read (RR) in IBM DB2 documentation. This isolation level corresponds with the ADO property set to adXactSerializable.

The isolation level can also be set by calling **ITransactionLocal::StartTransaction** with the appropriate value for the IsoLevel parameter to start a new transaction. Note that the same integer values used for the DBPROP_SESS_AUTOCOMMITISOLEVELS and DBPROP_SUPPORTEDTXNISOLEVELS property values are also used for the IsoLevel parameter passed to the **ITransactionLocal::StartTransaction** method. The legal values for the IsoLevel parameter are defined in TRANSACT.H while the OLE DB property values for isolation level are defined in OLEDB.h. While the #define macro strings used for the OLE DB property values and the IsoLevel parameter values differ, the integer values of these macros are the same.

The following table shows the OLE DB property values for isolation level and the equivalent IsoLevel parameter passed to **ITransactionLocal::StartTransaction**.

OLE DB Isolation Level Property	OLE DB IsoLevel Parameter
DBPROPVAL_TI_BROWSE	ISOLATIONLEVEL_BROWSE
DBPROPVAL_TI_CURSORSTABILITY	ISOLATIONLEVEL_CURSORSTABILITY
DBPROPVAL_TI_ISOLATED	ISOLATIONLEVEL_ISOLATED
DBPROPVAL_TI_READCOMMITTED	ISOLATIONLEVEL_READCOMMITTED
DBPROPVAL_TI_READUNCOMMITTED	ISOLATIONLEVEL_READUNCOMMITTED
DBPROPVAL_TI_REPEATABLEREAD	ISOLATIONLEVEL_REPEATABLEREAD
DBPROPVAL_TI_SERIALIZABLE	ISOLATIONLEVEL_SERIALIZABLE

Nested transactions are not supported by the OLE DB Provider for DB2. If there is already an active transaction on the session (that is, **StartTransaction** has been called with no matching **ITransaction::Commit** or **ITransaction::Abort**), it is not possible to start a new transaction below the current transaction. Calling **ITransactionLocal::StartTransaction** when there is already an active transaction on the session returns XACT_E_XTIONEXISTS.

IBM documents isolation level in DB2 documentation using somewhat different terms. The following table shows how the OLE DB values for isolation level are mapped to the terms used by IBM DB2 for isolation level.

OLE DB Isolation Level	IBM DB2 Isolation Level
DBPROPVAL_TI_BROWSE	Uncommitted Read (UR)
DBPROPVAL_TI_CURSORSTABILITY	Cursor Stability (CS)
DBPROPVAL_TI_ISOLATED	Repeatable Read (RR)
DBPROPVAL_TI_READCOMMITTED	Cursor Stability (CS)
DBPROPVAL_TI_READUNCOMMITTED	Uncommitted Read (UR)
DBPROPVAL_TI_REPEATABLEREAD	Read Stability (RS)
DBPROPVAL_TI_SERIALIZABLE	Repeatable Read (RR)

Transaction Support Using the OLE DB Provider for DB2

In earlier versions of the OLE DB provider for DB2 supplied with Microsoft SNA Server 4.0, the Auto commit property, an OLE DB provider-specific property, controlled whether work done through the provider (i.e. ICommand->Execute) was committed to the database as the work was done. This property defeated the intention of the OLE DB specification. Work done through the provider should be auto committed unless a request by the consumer is made to explicitly control when commits or aborts occur through the **ITransactionLocal**-object and the **StartTransaction** interface.

The OLE DB Provider for DB2 included with Microsoft Host Integration Server 2000 supports distributed transactions. This support is enabled by setting the provider specific property "UNITS OF WORK" to "DUW" (or checking the Distributed transactions option when configuring a UDL or DSN). The current DUW implementation does not have a notion of auto committing transactions so it is recommended to join the transaction through **ITransactionJoin**->**JoinTransaction** prior to performing work through **ICommandExecute** (or other interfaces). ADO applications do not generally join before starting their work, rather, they rely upon automatic transaction enlistment to enlist in the distributed transaction.

Using the earlier OLE DB Provider for DB2 supplied with SNA Server 4.0 Service Pack 3, RUW transaction state could cross OLE DB sessions on a single data source object. Because of this limitation, it was recommended that programmers utilize separate data source objects for each session object. To reduce the number of concurrent active sessions, then DBPROP_MULTIPLECONNECTIONS is set to VARIANT_FALSE.

The OLE DB Provider for DB2 included with Host Integration Server 2000 supports DBPROP_MULTIPLECONNECTIONS (the default value is VARIANT_TRUE) for remote unit of work (RUW) connections. This ensures that no transaction state conflicts across OLE DB session objects (ADO commands or recordsets on a single connection). This feature is a property of DBPROPSET_DATASOURCE. Therefore, you can only set this property after the OLE DB data source or ADO connection objects are created.

If the transaction has been auto enlisted, the default Transaction Isolation level is used (NC for the AS/400, CS for all other host platforms). If a join is requested (not applicable to ADO applications) after auto enlistment and work, then the Isolation Level specified on the join may differ from the default. This appears to be acceptable except in the case that a statement was prepared using one Isolation Level, the Isolation Level is then changed via **ITransactionJoin**, and then the prepared statement is executed again.

Stored Procedure Support Using the OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2 supports calling DB2 stored procedures. An application must use the CALL keyword before the SQL statement in order to execute a stored procedure. When using ADO, a CommandType property of adCmdStoredProc cannot be used for executing a stored procedure since ADO inserts an EXEC not CALL keyword before the command text. In order to execute a stored procedure using ADO, the **CommandType** property should be set to adCmdText and the CALL keyword should be used before the SQL statement containing the stored procedure to be executed.

When calling DB2 stored procedures, the following limitations apply when using the Microsoft OLE DB Provider for DB2:

- Binding output parameters of type REAL or DOUBLE is not supported.
- Calling stored procedures when the parameter values contain CHAR Mixed or GRAPHIC (DBCS) data types are not supported.
- Calling a non-existent procedure causes error.
- The OLE DB Provider for DB2 does not return single or multiple result sets.

Distributed Query Support Using the OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2 supports remote database access when configured as a linked server to Microsoft SQL Server™ using distributed queries. The distributed queries feature of SQL Server is sometimes referred to as the Distributed Query Processor (DQP). Microsoft SQL Server 2000 supports distributed queries to the OLE DB Provider for DB2 supplied with Microsoft Host Integration Server 2000 or Microsoft SNA Server 4.0 Service Pack 3 or later.

When using Microsoft SQL Server 2000 distributed queries and the OLE DB Provider for DB2 supplied with Host Integration Server or SNA Server 4.0, the following OLE DB provider options (displayed in the same order as in the SQL Server Enterprise Manager) are supported:

Provider Options in SQL Server 2000	**Comments**
Dynamic parameter	SQL Server will generate parameterized queries as an optimization for providers that support the '?' parameter marker syntax for parameterized queries in dynamic SQL. This option should not be enabled for the OLE DB Provider for DB2. .
Nested queries	SQL Server will generate nested queries for providers that support nested SELECT queries in the FROM clause. Some versions of DB2 have support for nested queries. This option should not be enabled for OLE DB Provider for DB2.
Level zero only	A level zero OLE DB provider is a very basic provider that does not support commands, and only level zero OLE DB interfaces are invoked against the provider. The Microsoft OLE DB Provider for DB2 is not a basic provider and uses commands. This option should not be enabled for the OLE DB Provider for DB2.
Allow InProcess	SQL Server allows the OLE DB provider to be instantiated as an in-process server. The default behavior is to instantiate the OLE DB provider outside the SQL Server process. Instantiating the provider outside the SQL Server process protects the SQL Server process from errors in the OLE DB provider. SQL Server requires an in-process server for handling specific types of data including long columns, text, and image data. The OLE DB Provider for DB2 does not currently support the DB2 Large Object (LOB) types. This option may be enabled or disabled for the OLE DB Provider for DB2, but this option is normally unnecessary when using SQL Server 2000.
Non transacted updates	SQL Server will disable support for transacted updates if this option is enabled. The Microsoft OLE DB Provider supports transactions, so this option is not appropriate. This option should not be enabled for the OLE DB Provider for DB2.
Index as access path	SQL Server will use the OLE DB Index object with OLE DB providers that support this feature. The OLE DB Provider for DB2 does not currently support the Index object, so this option is not appropriate. This option should not be enabled for the OLE DB Provider for DB2. .
Disallow adhoc accesses	SQL Server will use this option when only an ODBC driver is available. This option should not be enabled for the OLE DB Provider for DB2. .

The OLE DB provider options for managing distributed queries can be set using SQL Server Enterprise Manager. In the left pane of SQL Server Enterprise Manager, right-click a SQL Server instance and then select the **Security** tree to define a new linked server or change the properties of an existing linked server. Right click an existing linked server or create a new linked server. On the **General** tab, select the **Other data source** radio button and select the OLE DB Provider for DB2 for the **Provider name** from the dropdown listbox. Click the **Provider Options** button below the selected OLE DB provider to set the options for distributed queries. Check the appropriate checkboxes to enable an option for this linked server. Note that these options operate at the provider level. When the appropriate options are set for the OLE DB Provider for DB2, these settings apply to all linked server definitions using the same OLE DB Provider for DB2.

When using Microsoft SQL Server 7.0 and distributed queries with the OLE DB Provider for DB2 supplied with Host Integration Server 2000 or SNA Server 4.0, the **Allow InProcess** option must be enabled. This option is needed because SQL Server 7.0 will pass the proper authentication across the remote procedure call only when the OLE DB Provider for DB2 is configured for **Allow InProcess**. When creating a linked server for use with the OLE DB Provider for DB2 using SQL Server 7.0, you can use the

Microsoft SQL Server Enterprise Manager to configure the OLE DB options for linked servers. Configure the OLE DB Provider for DB2 to be loaded in-process (click the options button and check **Allow InProcess**). This will enable SQL Server 7.0 to initialize an instance of the OLE DB Provider for DB2 for distributed queries. Without the **Allow InProcess** option enabled, the user will receive the following error:

```
Server: Msg 7302, Level 16, State 1, Line 12; Could not create an instance of OLE DB provider
'DB2OLEDB'
```

The OLE DB Provider for DB2 performs data type and code page conversions on behalf of the OLE DB consumer application, in this case SQL Server's Distributed Query Processor. First, the provider will convert DB2 numeric and datetime data types to OLE DB numeric and datetime data types. The provider will do this on a best-match basis using information provided by DB2 in the DRDA SQL reply data structure for the result set. Second, the provider will convert the character data from the DB2 Coded Character Set Identifier (CCSID) to the Windows ANSI code page. For example in the case of an SQL SELECT fetch of data from DB2 for OS/400, the provider converts character data from EBCDIC to UNICODE and UNICODE to ANSI. The source EBCDIC CCSID value comes from the table column descriptor or from the DB2 database if the column CCSID is undefined. The target ANSI code page value comes from the value of the PC Code Page data source property. For example, to convert character data from CCSID 1026, IBM EBCDIC Turkish (Latin-5), to ANSI code page 1254, Turkish, then the PC CodePage would need to be set to 1254. OLE DB consumers may convert the data once again from OLE DB to some native data type. In this case, the OLE DB consumer application is distributed query processor. When using distributed queries, SQL Server does perform numeric and datetime conversions from OLE DB to SQL Server data types.

UNIONs in SELECT statements are not supported by the current version of the OLE DB Provider for DB2 when used with distributed queries. For example, the following SELECT statement will fail:

```
SELECT * FROM ( SELECT TITLE_ID FROM SNA.TITLE )
```

The SQL parser built into the OLE DB Provider for DB2 does not properly parse these UNION statements in a way compatible with DB2. The above statement will generate a type 199 error "Keyword FROM not expected. Valid Tokens: LEFT CROSS INNER EXCEPTION." When performing the same query with correlation names added, the error becomes a type 104 error "Token was not valid. Valid tokens: LEFT CROSS INNER EXCEPTION."

Linked server definitions can also be created or deleted using stored procedures as well as through the SQL Server Enterprise Manager.

When creating linked server definitions, the @catalog parameter of the sp_addlinkedserver procedure corresponds to the OLE DB provider-specific Initial Catalog property. Additionally, when creating linked server definitions, one can use the contents of a UDL for the @provstr parameter value or enter the short provider string keyword arguments (see the ADO ConnectionString property for details), which are consumed via the OLE DB DBPROP_INIT_PROVIDERSTRING (Extended Properties) property.

The sample below illustrates how to create a linked server for DB2/MVS using an SNA Connection.

```
USE master
GO
EXEC sp_dropserver 'DB2MVS_SNA', 'droplogins'
GO
EXEC sp_addlinkedserver
    @server = 'DB2MVS_SNA',
    @srvproduct = 'Microsoft OLE DB Provider for DB2',
    @provider = 'DB2OLEDB',
    @catalog = 'P390D37',
    @provstr='InitCat=P390D37;NetLib=SNA;LOCALLU=MVSRUS;
       REMOTELU=P390L37;MODENAME=IBMRDB;PkgCol=MSPKG;DefSch=DB2DEMO'
GO
EXEC sp_addlinkedsrvlogin
    @rmtsrvname='DB2MVS_SNA',
    @useself=false,
    @locallogin=NULL,
    @rmtuser='wnw999',
    @rmtpassword='wnw999'
GO
```

The sample below illustrates how to create a linked server for DB2/MVS using a TCP/IP Connection.

```
USE master
GO
EXEC sp_dropserver 'DB2MVS_IP', 'droplogins'
GO
EXEC sp_addlinkedserver
    @server = 'DB2MVS_IP',
    @srvproduct = 'Microsoft OLE DB Provider for DB2',
    @provider = 'DB2OLEDB',
    @catalog = 'P390D37',
    @provstr='InitCat=P390D37;NetLib=TCPIP;NetAddr=MVSrUS;
        NetPort=446;PkgCol=MSPKG;DefSch=DB2DEMO'
GO
EXEC sp_addlinkedsrvlogin
    @rmtsrvname='DB2MVS_IP',
    @useself=false,
    @locallogin=NULL,
    @rmtuser='wnw999',
    @rmtpassword='wnw999'
GO
```

The sample below illustrates several DB2 linked server queries using a TCP/IP Connection.

```
/* SELECT using four-part linked server query:   */
/*    <Linked Server>.<Catalog>.<Schema>.<Table> */
SELECT * FROM DB2MVS_IP.P390D37.DB2DEMO.DEPARTMENT
SELECT DEPTNAME FROM DB2MVS_IP.P390D37.DB2DEMO.DEPARTMENT WHERE DEPTNO = 'A00'

/* SELECT using pass-through OPENQUERY with      */
/*     three-part naming convention              */
SELECT * FROM OPENQUERY(DB2MVS_IP,"SELECT * FROM P390D37.DB2DEMO.EMP_ACT")

/* SELECT using pass-through OPENROWSET with     */
/*     two-part naming convention                */
SELECT * FROM OPENROWSET
    (
    'DB2OLEDB',
    'InitCat=P390D37;NetLib=TCPIP;NetAddr=MVSrUS;NetPort=446;
         PkgCol=MSPKG;DefSch=DB2DEMO;User ID=WNW999;Password=WNW999',
    'SELECT * FROM DB2DEMO.EMPLOYEE'
    )
```

The sample below illustrates several SQL Server Views to access a DB2 linked server using a TCP/IP Connection.

```
/* Create SQL Server View using DB2 linked server query */
USE DB2Demo
GO
DROP VIEW DB2VIEW
GO
CREATE VIEW DB2View
AS
SELECT ORD_ID, ORDERDATE, ORDERSTATUS
    FROM DB2MVS_IP.P390D37.DB2DEMO.ORDERS
GO

/* Access Db2 data using SQL Server View */
SELECT * FROM DB2Demo.dbo.DB2View
```

The sample below illustrates the SQL SELECT, UPDATE, and DELETE commands using four-part linked server queries to DB2 over a TCP/IP Connection.

```
SELECT * FROM DB2MVS_IP.P390D37.DB2DEMO.CUSTOMERS
INSERT INTO DB2MVS_IP.P390D37.DB2DEMO.CUSTOMERS VALUES (1002,
    'password', 'User', 'DB2Demo', 'One Microsoft Way', 'Redmond',
    '425-882-8080', 'WA', '98052', 'mssna@microsoft.com')
```

```
UPDATE DB2MVS_IP.P390D37.DB2DEMO.CUSTOMERS  SET PHONE = '206-882-8080'
    WHERE CUST_ID = 1002
DELETE FROM DB2MVS_IP.P390D37.DB2DEMO.CUSTOMERS WHERE CUST_ID = '1002'
```

The sample below illustrates invoking DB2 linked server queries from SQL Server stored procedure.

```
USE DB2Demo
DROP PROCEDURE spGetOrderDetails_DB2MVS_IP

CREATE PROCEDURE spGetOrderDetails_DB2MVS_IP
    @ORDER_ID INT
AS
BEGIN
    SELECT a.*, b.Title, b.SubTitle, b.Author, b.ISBN, b.Weight
        FROM DB2MVS_IP.P390D37.DB2DEMO.ORDERDETAILS AS a
    INNER JOIN Titles AS b
        ON a.Prod_Id = b.TitleId
        WHERE a.Ord_Id >= @ORDER_ID
END

EXEC spGetOrderDetails_DB2MVS_IP 1001
```

Note: for more information on the above stored procedures, see the Microsoft SQL Server books online.

INSERT, UPDATE and DELETE statements when using four-part linked server queries will invoke the client cursor engine (CCE). This means that some statements may fail or update incorrect columns. For example, if there is not a unique key column on the target tables or there are not enough unique values for the CCE to accurately guess which columns to update. For INSERT, UPDATE and DELETE linked server queries, there must be either a unique index or unique values.

Using distributed queries, SQL Server will return all rows for the target table locally, then sort through results using the client cursor engine to find a unique value (composed of all columns if need be). If the CCE can't find a unique value, then the distributed query will fail to perform the INSERT, UPDATE, or DELETE statement.

Using the OLE DB Provider for DB2, SQL Server distributed queries return the incorrect precision for DECIMAL data types when run OPENQUERY. The correct precision is returned for DECIMAL data types with using a linked server query.

Query Designer Support Using the OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2 supports using the Query Designer that is included with Microsoft SQL Server. When Query Designer is launched from products such as SQL Server Data Transformation Services (DTS), the OLE DB Provider for DB2 is initialized with no provider-specific properties and an invalid property error may be generated.

To work around this problem, create a data link with the provider-specific properties required to connect to the host. These properties are specified in the Extended Properties field on the All tab when configuring a data source. Typically, the remote LU Alias, the local LU Alias, and the Default Schema should be specified using either the full property name (APPC Remote LU Alias=REMLU;APPC Local LU Alias;Default Schema=MYLIB) or the short names (RLU=REMLU;LLU;DEFSCH=MYLIB).

Other properties, such as APPC Mode Name, may need to be specified if the default values for that property are not acceptable. You should also specify a Default Schema. If a Default Schema is not specified, Query Designer will display all schemas on the remote system (including libraries, collections, etc.), which can result in slow performance.

Data Transformation Services Support Using the OLE DB Provider for DB2

The Microsoft OLE DB Provider for DB2 supports using the Data Transformation Services (DTS) facility provided with Microsoft SQL Server. DTS can be used for file transfer between DB2 and Microsoft SQL Server. There are some special considerations necessary when using DTS and the OLE DB Provider for DB2.

A DB2/400 database member can only be accessed in a SQL statement by using an ALIAS or SYNONYM. For example, if you have several members of a SALES.REVENUE database file, each member storing monthly sales information, then you would access one month's data member by using an ALIAS. First, you must create the ALIAS.

```
CREATE ALIAS SALES.REVENUE_JANUARY FOR SALES.REVENUE(JANUARY)
```

Second, you must use this alias when issuing a SQL statement. You can access individual members using an ALIAS in the following SQL statements: SELECT, SELECT INTO, INSERT, UPDATE, and DELETE.

```
SELECT * FROM SALES.REVENUE_JANUARY
```

Once the alias is created, you should be able to see the table listed in the systables by issuing the following SQL statement:

```
SELECT * FROM QSYS2.SYSTABLES WHERE TABLE_SCHEMA = 'SALES'
```

The REVENUE_JANUARY table type is "ALIAS". Many OLE DB providers and ODBC drivers do not list ALIASes as TABLEs when returning catalog schema. Note that DTS will not list the table when using the Microsoft OLE DB Provider for DB2 or the Microsoft ODBC Driver for DB2. If the ALIAS is not listed, then simply utilize the Query option in DTS to specify a SELECT * FROM SALES.REVENUE_JANUARY statement.

Note that it is possible to use the OVRDBF CL command. However, using an override database file has significant overhead, whereas ALIASes are less demanding on DB2 server resources. Additionally, the OVRDBF command must be issued each time (likely as part of a stored procedure), whereas an ALIAS is persistent.

For more information, see the DB2/400 IBM SQL Reference Version 4, Document Number SC41-5612-02.

Microsoft SQL Server's Data Transformation Services will not automatically create target DB2 tables that can store DBCS or mixed DBCS-SBCS data. When using DTS to move DBCS or mixed data to DB2 as a source, then one of the following techniques are recommended:

- Create the target DB2 table ahead of time.
- Edit the CREATE TABLE script within the DTS package to include the required CCSID and FOR MIXED clauses as appropriate for the target DB2 platform and version.

Exporting DBCS VARCHAR data from SQL Server using DTS is not currently supported by the Microsoft OLE DB Provider for DB2.

When importing TIME data fields into Microsoft SQL Server using DTS and the OLE DB Provider for DB2, the second field is incorrectly set to a value of '00'. This behavior occurs when the field data type in SQL Server is Smalldatetime.

SQL Server Replication Using the OLE DB Provider for DB2

Host Integration Server 2000 supports using the Microsoft OLE DB Provider for DB2 and Microsoft SQL Server 7.0 Replication.

In order to use the Microsoft OLE DB Provider for DB2 with Microsoft SQL Server 7.0 Replication, the following software must the installed:

- Microsoft Host Integration Server 2000
- Microsoft SQL Server 7.0 with Service Pack 2 or later
- Microsoft SQL Server 7.0 with Service Pack 1 with the appropriate hotfix

When using transactional replication, object names, such as tables and constraints, can be longer in Microsoft SQL Server than in DB2. Therefore, when moving data from Microsoft SQL Server to DB2, one of the following strategies should be used for compatibility:

- Create the Microsoft SQL Server tables and constraints so that the names do not exceed the limits of DB2 (see the DB2 SQL Reference for the target platform and version of DB2).
- Create the Microsoft SQL Server tables and constraints with names up to the maximum supported by Microsoft SQL Server, then run ALTER TABLE SQL scripts against the Microsoft SQL Server distribution database after replication has created the publication, but before creating the subscriptions.

In order for replication to work properly, SQL Server-to-DB2 data type mapping entries must be provided for several data types. These data type mappings must be stored in the Microsoft SQL Server MSdatatype_mappings table in the msdb database. Data type mapping entries must be provided to properly support the CHAR and VARCHAR data types when used to store SBCS character strings or binary data (BINARY and VARBINARY) with replication. The mappings for CHAR and VARCHAR are needed due to the limits for the boundary values using the OLE DB Provider for DB2. The mappings for binary data types are required since binary data is stored as character data using the OLE DB Provider for DB2.

This SQL Server msdb.MSdatatype_mappings table can be updated using a stored procedure that comes with SQL Server. The exec sp_add_datatype_mapping command is a stored procedure that comes with the SQL Server. Users should run these data mappings in the SQL Query Analyzer (isqlw.exe). Each line of the exec sp_add_datatype_mapping procedure will add one data mapping to msdb.Msdatatype_mappings.

The Microsoft OLE DB Provider for DB2 when accessing DB2 on all platforms and versions does not support the following Microsoft SQL Server data types with direct mappings. The following is a summary of the limitations.

Microsoft OLE DB Provider for DB2 does not support DB2 GRAPHIC, VARGRAPHIC and LONG VARGRAPHIC data types. Mappings to these DB2 types should be avoided.

Microsoft OLE DB Provider for DB2 does not support DB2 Large Object data types, such as CLOB, DBCLOB, and BLOB, which are used in DB2 Universal Database. Microsoft OLE DB Provider for DB2 does not support the DB2 BIGINT data types, which is used in DB2 Universal Database for Windows NT.

Microsoft SQL Server NCHAR, NVARCHAR, NTEXT, and SYSNAME data types do not map well to supported DB2 data types. The Microsoft SQL Server TIMESTAMP data type should be mapped to a DB2 CHAR FOR BIT DATA data type. The TIMESTAMP data type in SQL Server is used only for the purposes of table logging and not for end user data. Therefore, you should not map the SQL Server TIMESTAMP data type to the DB2 TIMESTAMP.

The sample mapping table entries map Microsoft SQL Server TEXT and IMAGE to DB2 VARCHAR and DB2 VARCHAR FOR BIT DATA respectively. These Microsoft SQL Server data types can represent up to 2GB of data. In contrast, the DB2 VARCHAR data types can hold at most approximately 32k of data. As such, data truncation may occur.

For Microsoft SQL Server data types, MONEY, SMALLMONEY, DECIMAL, NUMERIC, data values can be replicated to DB2 correctly, but these floating-point data types are mapped to STRING instead of FLOAT. Once these data values are replicated to DB2, the string values should be converted back to float before performing calculations. An alternative is to do the calculations and update using SQL Server before replicating these data values over to DB2.

The following commands will create the appropriate data type mapping entries.

SQL Replication with DB2 for OS/400

When used with DB2 for OS/400, run the following commands in a stored procedure:

```
exec dbo.sp_add_datatype_mapping 'DB2/400', 'bit',
    'SMALLINT', 1,  0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'char', 'CHAR', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'datetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'decimal',
    'DECIMAL', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'double precision',
    'DOUBLE', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'float', 'FLOAT', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'int', 'INT', 10, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'money',
    'DECIMAL', 19, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'numeric',
    'NUMERIC', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'real', 'REAL', 24, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'smalldatetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'smallint',
    'SMALLINT', 5, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'smallmoney',
    'DECIMAL', 10, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'text',
    'VARCHAR', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'tinyint',
    'SMALLINT', 1, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'uniqueidentifier',
    'CHAR', 38, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'varchar',
    'VARCHAR', 8000, 4, 1
```

When using Microsoft SQL Server 7.0 (including SQL Server 7.0 with Service Pack 1 or Service Pack 2), replication does not support the following data conversions:

```
exec dbo.sp_add_datatype_mapping 'DB2/400', 'binary',
    'CHAR () FOR BIT DATA', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'image',
    'VARCHAR () FOR BIT DATA', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'timestamp',
    'CHAR () FOR BIT DATA', 8, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/400', 'varbinary',
    'VARCHAR () FOR BIT DATA', 8000, 4, 1
```

Microsoft SQL Server 7.0 with Service Pack 3 is required to support replication with these data conversions.

SQL Replication with DB2 for MVS

When used with DB2 for MVS , run the following commands:

```
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'bit',
    'SMALLINT', 1,  0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'char', 'CHAR', 254, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'datetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'decimal',
    'DECIMAL', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'double precision',
    'DOUBLE', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'float', 'FLOAT', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'int',
    'INT', 10, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'money',
    'DECIMAL', 19, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'numeric',
    'NUMERIC', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'real', 'REAL', 24, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'smalldatetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'smallint',
    'SMALLINT', 5, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'smallmoney',
   'DECIMAL', 10, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'text',
   'VARCHAR', 4045, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'tinyint',
    'SMALLINT', 1, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'uniqueidentifier',
    'CHAR', 38, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'varchar',
    'VARCHAR', 4045, 4, 1
```

When using Microsoft SQL Server 7.0 (including SQL Server 7.0 with Service Pack 1 or Service Pack 2), replication does not support the following data conversions:

```
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'binary',
    'CHAR () FOR BIT DATA', 254, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'image',
   'VARCHAR () FOR BIT DATA', 4045, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'timestamp',
   'CHAR () FOR BIT DATA', 8, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/MVS', 'varbinary',
    'VARCHAR () FOR BIT DATA', 4045, 4, 1
```

Microsoft SQL Server 7.0 with Service Pack 3 is required to support replication with these data conversions.

SQL Replication with DB2 UDB for AIX

When used with DB2 Universal Database for AIX, run the following commands:

```
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'bit',
    'SMALLINT', 1,  0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'char',
    'CHAR', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'datetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'decimal',
    'DECIMAL', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'double precision',
    'DOUBLE', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'float',
    'FLOAT', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'int', 'INT', 10, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'money',
    'DECIMAL', 19, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'numeric',
    'NUMERIC', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'real', 'REAL', 24, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'smalldatetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'smallint',
    'SMALLINT', 5, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'smallmoney',
    'DECIMAL', 10, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'text',
    'VARCHAR', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'tinyint',
    'SMALLINT', 1, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'uniqueidentifier',
    'CHAR', 38, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'varchar',
    'VARCHAR', 8000, 4, 1
```

When using Microsoft SQL Server 7.0 (including SQL Server 7.0 with Service Pack 1 or Service Pack 2), replication does not support the following data conversions:

```
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'binary',
    'CHAR () FOR BIT DATA', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'image',
    'VARCHAR () FOR BIT DATA', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'timestamp',
    'CHAR () FOR BIT DATA', 8, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/6000', 'varbinary',
    'VARCHAR () FOR BIT DATA', 8000, 4, 1
```

Microsoft SQL Server 7.0 with Service Pack 3 is required to support replication with these data conversions.

SQL Replication with DB2 UDB for NT

When used with DB2 Universal Database for NT, run the following commands:

```
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'bit',
    'SMALLINT', 1,  0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'char', 'CHAR', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'datetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'decimal',
    'DECIMAL', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'double precision',
    'DOUBLE', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'float', 'FLOAT', 53, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'int', 'INT', 10, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'money',
    'DECIMAL', 19, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'numeric',
    'NUMERIC', 31, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'real', 'REAL', 24, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'smalldatetime',
    'TIMESTAMP', 26, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'smallint',
    'SMALLINT', 5, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'smallmoney',
    'DECIMAL', 10, 3, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'text',
    'VARCHAR', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'tinyint',
    'SMALLINT', 1, 0, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'uniqueidentifier',
    'CHAR', 38, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'varchar',
    'VARCHAR', 8000, 4, 1
```

When using Microsoft SQL Server 7.0 (including SQL Server 7.0 with Service Pack 1 or Service Pack 2), replication does not support the following data conversions:

```
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'binary',
    'CHAR () FOR BIT DATA', 8000, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'image',
    'VARCHAR () FOR BIT DATA', 32739, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'timestamp',
    'CHAR () FOR BIT DATA', 8, 4, 1
exec dbo.sp_add_datatype_mapping 'DB2/NT', 'varbinary',
    'VARCHAR () FOR BIT DATA', 8000, 4, 1
```

Microsoft SQL Server 7.0 with Service Pack 3 is required to support replication with these data conversions.

Code Page Support Using the OLE DB Provider for DB2

When creating data links for use with the OLE DB Provider for DB2, the Host CCSID (character code set identifier) should be configured in the data source to match the DB2 data as represented on the remote host computer. The Host CCSID parameter defaults to EBCDIC U.S./Canada (37) when using the OLE DB Provider for DB2.

Depending on the version of Windows being used, to support specific code page conversions, you may need to install the appropriate National Language Support (NLS) file for your locale.

On Windows 2000, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows 2000.

On Windows NT 4.0, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows NT or when you install the Windows NT Language Pack on a non-localized version of Windows NT. The Windows NT Language Pack is available on the Windows NT 4.0 CD-ROM in the LANGPACK directory. You install the locale components of the language pack as needed by either making a change in the Control Panel Locales applet or by installing one of the locale-specific INF files.

On Windows 98 and Windows 95, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows 98 or Windows 95.

The following sections discuss the character code set identifiers (CCSIDs) supported by OLE DB Provider for DB2 in Host Integration Server 2000.The The tables in these sections list the INF files by name that are required under Windows NT 4.0 for a specific codepage (european.inf, for example). Typically you would install locales one at a time as needed.

ANSI Code Page Support Using the OLE DB Provider for DB2

IBM DB2 Universal Database for Windows NT and IBM DB2 Universal Database for AIX are frequently configured to use ANSI code pages, for example ANSI 1253 (Greek). Host Integration Server 2000 includes support for some ANSI code pages for purposes of ANSI-to-UNICODE-to-ANSI conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These ANSI code pages can be used when accessing IBM DB2 Universal Database on Windows NT and IBM DB2 ON AIX (not all of these ANSI code pages are supported on IBM DB2 Universal Database for AIX).

The following table shows the ANSI character code set identifiers (CCSIDs) supported by OLE DB Provider for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
ANSI - Arabic	1256	1256	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
ANSI - Baltic	1257	1257	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ANSI - Cyrillic	1251	1251	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
ANSI - Central Europe	1250	1250	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ANSI - Greek	1253	1253	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
ANSI - Hebrew	1255	1255	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
ANSI - Latin I	1252	1252	Support for this codepage is normally installed as part of the operating system on Windows 2000, Windows NT, Windows 98, and Windows 95. On Windows NT 4.0, support for this NLS Code Page is installed using the us_eng.inf file from the Language Pack.
ANSI - Turkish	1254	1254	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
ANSI/OEM - Japanese Shift JIS	932	932	On Windows NT 4.0, support for this NLS Code Page is installed using the japanese.inf file from the Language Pack.
ANSI/OEM - Korean	949	949	On Windows NT 4.0, support for this NLS Code Page is installed using the korean.inf file from the Language Pack.
ANSI/OEM - Simplified Chinese GBK	936	936	On Windows NT 4.0, support for this NLS Code Page is installed using the exchsrvr.inf file from the Language Pack.
ANSI/OEM - Thai	874	874	On Windows NT 4.0, support for this NLS Code Page is installed using the thai.inf file from the Language Pack.
ANSI/OEM - Traditional Chinese Big5	950	950	On Windows NT 4.0, support for this NLS Code Page is installed using the tchinese.inf file from the Language Pack.
ANSI/OEM - Viet Nam	1258	1258	On Windows NT 4.0, support for this NLS Code Page is installed using the vietnam.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows 2000 or Windows NT definitions for these NLS files. The Microsoft NLS Code Page column represents the code page number that is registered and associated with an ANSI-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the OLE DB Provider for DB2. When setting the Host CCSID or PC Code Page attribute/property using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the ANSI code page in IBM publications, which for these supported ANSI CCSIDs are the same as the Microsoft CCSID values. IBM lists their ANSI support in publications by referencing the display name which for these ANSI code pages is the same as the Microsoft display name. The OLE DB Provider for DB2 does not recognize or display the IBM CCSID values when configuring data sources using data links. The OLE DB Provider for DB2 maps the Microsoft NLS numbers to ANSI NLS files which correspond with the appropriate IBM CCSID numbers. The OLE DB Provider for DB2, as well as Microsoft ODBC Driver for DB2, pass the corresponding IBM CCSID to the DB2 system at run time even though you configure the provider or driver to use the Microsoft NLS number.

These are the only ANSI pages currently supported by the OLE DB Provider for DB2 in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. IBM supports additional ANSI pages, however, the ANSI code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM ANSI code pages (CCSIDs) match.

EBCDIC Code Page Support Using the OLE DB Provider for DB2

IBM DB2 for MVS, IBM DB2 for OS/390, and IBM DB2 for OS/400 are frequently configured to use EBCDIC code pages, for example EBCDIC 875 (Greek Modern). Host Integration Server 2000 includes support for most EBCDIC code pages for purposes of EBCDIC-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-EBCDIC, and EBCDIC-to-UNICODE-to-EBCDIC conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These EBCDIC code pages can be used when accessing IBM DB2 on a variety of platforms (not all of these EBCDIC code pages are supported on all versions of IBM DB2).

The following table shows the EBCDIC character code set identifiers (CCSIDs) supported by OLE DB Provider for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
IBM EBCDIC - Arabic	20420	420	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
IBM EBCDIC - Cyrillic (Russian)	20880	880	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
IBM EBCDIC - Cyrillic (Serbian, Bulgarian)	21025	1025	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
IBM EBCDIC - Denmark/Norway	20277	277	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Denmark/Norway (Euro)	1142	1142	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Finland/Sweden	20278	278	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Finland/Sweden (Euro)	1143	1143	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - France	20297	297	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - France (Euro)	1147	1147	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Germany	20273	273	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Germany (Euro)	1141	1141	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Greek	20423	423	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
IBM EBCDIC - Greek (Modern)	875	875	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
IBM EBCDIC - Hebrew	20424	424	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
IBM EBCDIC - Icelandic	20871	871	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Icelandic (Euro)	1149	1149	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - International	500	500	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - International (Euro)	1148	1148	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Italy	20280	280	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Italy (Euro)	1144	1144	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Japan English/Kanji (Extended)	939	939	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japan English/Kanji (Extended)	5035	5035	Support for this double-byte character set is supplied using TRNSDT.

IBM EBCDIC - Japan Katakana/Kanji (Extended)	930	930	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japan Katakana/Kanji (Extended)	5026	5026	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japanese	931	931	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Korea (Extended)	933	933	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Latin America/Spain	20284	284	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Latin America/Spain (Euro)	1145	1145	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Multilingual/ROECE (Latin-2)	870	870	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Simplified Chinese (Extended)	935	935	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Thai	20838	838	On Windows NT 4.0, support for this NLS Code Page is installed using the thai.inf file from the Language Pack.
IBM EBCDIC - Traditional Chinese (Extended)	937	937	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Turkish (Latin-3)	20905	905	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
IBM EBCDIC - Turkish (Latin-5)	1026	1026	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
IBM EBCDIC - U.S./Canada	037	37	On Windows NT 4.0, support for this NLS Code Page is installed using the us_eng.inf file from the Language Pack.
IBM EBCDIC - U.S./Canada (Euro)	1140	1140	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - United Kingdom	20285	285	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - United Kingdom (Euro)	1146	1146	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows 2000 or Windows NT definitions for these NLS files. The Microsoft NLS Code Page column represents the code page number that is registered and associated with an EBCDIC-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the OLE DB Provider for DB2. When setting the Host CCSID or PC Code Page attribute/property using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the EBCDIC code page in IBM publications. IBM lists their EBCDIC support in publications by referencing the display name which for these EBCDIC code pages is the same as the Microsoft display name. The OLE DB Provider for DB2 does not recognize or display the IBM CCSID values when configuring data sources using data links. The OLE DB Provider for DB2 maps the Microsoft NLS numbers to EBCDIC NLS files which correspond with the appropriate IBM CCSID numbers. The OLE DB Provider for DB2, as well as Microsoft ODBC Driver for DB2, pass the corresponding IBM CCSID to the DB2 system at run time even though you configure the provider or driver to use the Microsoft NLS number.

These are the only EBCDIC pages currently supported by the OLE DB Provider for DB2 in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. IBM supports additional EBCDIC pages, however, the EBCDIC code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM EBCDIC code pages (CCSIDs) match.

ISO Code Page Support Using the OLE DB Provider for DB2

IBM DB2 Universal Database for Windows NT and IBM DB2 Universal Database for AIX are frequently configured for an ISO code page, for example ISO 819 (Latin I). Host Integration Server 2000 includes support for some ISO code pages for purposes of ISO-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-ISO, and ISO-to-UNICODE-to-ISO conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These ISO code pages can be used when accessing IBM DB2 Universal

The following table shows the ISO character code set identifiers (CCSIDs) supported by OLE DB Provider for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
ISO 8859-1 Latin 1	28591	819	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-2 Central Europe	28592	912	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-5 Cyrillic	28595	915	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
ISO 8859-6 Arabic	28596	1089	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
ISO 8859-7 Greek	28597	813	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
ISO 8859-8 Hebrew	28598	916	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
ISO 8859-9 Turkish	28599	920	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 6937 Non-Spacing Accent	20269	819	Note that ISO 6937 (CCSID 20269) is not supported by the OLE DB Provider for DB2, but is displayed in the list of configuration options when creating or modifying data sources.
ISO 8859-15 Latin 9 (Euro)	20865	923	NLS Code Page 819 with support for the Euro. On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows NT Language Pack definitions for these NLS files.

The Microsoft NLS Code Page column represents the code page number that is registered and associated with an ISO-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the OLE DB Provider for DB2. When setting the Host CCSID or PC Code Page attribute/property using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the ISO code page in IBM publications. IBM lists their ISO support in publications by referencing the locale name (Bulgaria for ISO8859-5 and 915, for example) rather than simply using ISO 8859-5 Cyrillic as used by Microsoft. The OLE DB Provider for DB2 does not recognize or display the IBM CCSID values when configuring data sources using data links. The OLE DB Provider for DB2 maps the Microsoft NLS numbers to ISO NLS files which correspond with the appropriate IBM CCSID numbers. The OLE DB Provider for DB2, as well as Microsoft ODBC Driver for DB2, pass the corresponding IBM CCSID to the DB2 system at run time even though you configure the provider or driver to use the Microsoft NLS number.

Note that IBM CCSID 819 is associated with both ISO 8859-1 Latin 1 and ISO 6937 Non-Spacing Accent. It is up to the user to choose the standard ISO 8859-1 Latin 1 code page by selecting NLS code page 28591 or the modified code page ISO 6937 Non-Spacing Accent by selecting NLS code page 20269. Note that ISO 6937 Non-Spacing Accent (CCSID 20269) is not currently supported by the OLE DB Provider for DB2, but is displayed in the configuration options when creating or modifying data sources.

IBM CCSID 916 (ISO 8859-8) supports Hebrew "visual sort order". IBM CCSID 920 (ISO 8859-8 derivation) supports Hebrew "logical sort order". Although Microsoft supports the logical sort order with NLS 38598, this NLS file is only distributed with Internet Explorer 5 or Windows 2000. The OLE DB Provider for DB2 has not been tested using the ISO 8859-8 derivation matching IBM CCSID 920 and does not support this configuration.

These are the only ISO pages currently supported in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. Microsoft supports a number of additional ISO pages. IBM also supports additional ISO pages. However, the code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM CCSIDs match.

DBCS Code Page Support Using the OLE DB Provider for DB2

Support for Double-Byte Character String (DBCS) data is limited using the OLE DB Provider for DB2. Conversions between DBCS and ANSI code pages are not supported. Conversions between DBCS and ISO code pages are not supported. Positioned updates against DBCS EBCDIC implementations of DB2 are not supported.

The DB2 GRAPHIC data types (GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC) are not supported. These DB2 data types support DBCS (not mixed) data. Mixed data types are supported using CHAR FOR MIXED DATA, VARCHAR FOR MIXED DATA, and LONGVARCHAR FOR MIXED DATA.

Parameterized SQL statements or calling stored procedures when the parameter values contain Mixed or DBCS characters are not supported.

Data Conversion Using the OLE DB Provider for DB2

The design of the OLE DB APIs is similar to the APIs provided by ODBC and other ISAM APIs. The APIs are handle-based. After opening a file, the application can determine the buffer size required to store a row, use the cursor APIs to move, and optionally retrieve one or more rows of data using the row-level binding.

Data is converted to default C data types as defined in ODBC and OLE DB illustrated in the following table:

DB2 data type	Default C data type	Comments
BIGINT		An eight-byte integer. This data type is converted to DBTYPE_I8 for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.
BLOB		A Binary Large Object (BLOB) is a varying-length string that can be up to 2 gigabytes in length. A BLOB is primarily intended to hold binary data. This data type is converted to a DBTYPE_STR for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.
CHAR (Bit)	char string[]	A fixed length string. This data type is converted to a DBTYPE_BSTR for use by OLE DB.
CHAR (SBCS)	char string[]	A fixed-length SBCS character string. This data type is converted to a DBTYPE_BSTR for use by OLE DB.
CHAR (Mixed Data)	char string[]	A fixed-length mixed character string. This data type is converted to a DBTYPE_BSTR for use by OLE DB.
CLOB		A Character Large Object (CLOB) is a varying-length string that can be up to 2 gigabytes in length. A CLOB is used to store large single-byte character set data. A CLOB is considered to be a character string. This data type is converted to a DBTYPE_STR for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.
DATE	date struct	A ten byte date string. This data type is converted to a DBTYPE_DATE for use by OLE DB.
DEC	unsigned char number[]	A packed decimal number. This data type is converted to a DBTYPE_DECIMAL for use by OLE DB.
DOUBLE	double	An 8-byte double-precision floating point number. This data type is converted to a DBTYPE_R8 for use by OLE DB.
FLOAT	double	An 8-byte double-precision floating point number. This data type is the same as a DOUBLE. This data type is converted to a DBTYPE_R8 for use by OLE DB.
GRAPHIC (DBCS)	unsigned char binary[]	A fixed-length graphic string consisting of a sequence of double byte character string (DBCS) data. This data type is converted to a DBTYPE_WSTR for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.
INTEGER	int	A four-byte integer ranging in value from -2,147,463,648 to +2,147,483,647. This data type is converted to a DBTYPE_I4 for use by OLE DB.

LONG VARCHAR (Bit)	char string[]	A varying-length character string up to 32,740 characters in length. This data type is converted to a DBTYPE_STR for use by OLE DB.
LONG VARCHAR (SBCS)	char string[]	A varying-length SBCS character string up to 32,740 characters in length. This data type is converted to a DBTYPE_STR for use by OLE DB.
LONG VARCHAR (Mixed)	char string[]	A varying-length mixed-character string. This data type is converted to a DBTYPE_STR for use by OLE DB.
LONG VARGRAPHIC (DBCS)	unsigned char binary[]	A varying-length graphic string consisting of a sequence of double byte character string (DBCS) data ranging up to 16,383 DBCS characters in length. This data type is converted to a DBTYPE_WSTR for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.
SMALLINT	short	A SMALLINT (small integer) is a two-byte integer with a precision of 5 digits ranging from -32,768 to +32,767. This data type is converted to a DBTYPE_I2 for use by OLE DB.
REAL	float	A 4-byte single-precision floating point number. This data type is converted to a DBTYPE_R4 for use by OLE DB.
TIME	time struct	An 8-byte time string. This data type is converted to a DBTYPE_TIME for use by OLE DB. When using ActiveX Data Objects to return data from a DB2 TIME data type, ADO returns a DATETIME value.
TIMESTAMP	timestamp struct	A 26-byte string representing the date, time, and microseconds. This data type is converted to a DBTYPE_DBTIMESTAMP for use by OLE DB.
VARCHAR (Bit)	char string[]	A varying-length character string. The maximum length of the string is dependent on the version and the platform that DB2 is running on. This data type is converted to a DBTYPE_STR for use by OLE DB.
VARCHAR (SBCS)	char string[]	A varying-length character string. The maximum length of the string is dependent on the version and the platform that DB2 is running on. This data type is converted to a DBTYPE_STR for use by OLE DB.
VARCHAR (Mixed)	char string[]	A varying-length character string. The maximum length of the string is dependent on the version and the platform that DB2 is running on. This data type is converted to a DBTYPE_STR for use by OLE DB.
VARGRAPHIC (DBCS)	unsigned char binary[]	A varying-length graphic string consisting of a sequence of double byte character string (DBCS) data. The maximum length of the string is dependent on the version and the platform that DB2 is running on. This data type is converted to a DBTYPE_WSTR for use by OLE DB. This data type is not supported by the OLE DB Provider for DB2.

Note that the maximum length of fixed-length CHAR, fixed-length GRAPHIC, VARCHAR, and VARGRAPHIC data types is dependent on the version of DB2 that is being accessed. For example, the maximum length of the CHAR data type on DB2 for OS/390 is 254 characters, while the maximum length of this same host data type is 32,765 on DB2/400.

Data conversions from a large numeric type to a small numeric type are supported (from DOUBLE to SINGLE and from INT to SMALLINT, for example), however truncation and conversion errors can occur that will not be reported by the OLE DB Provider for DB2.

Note that the OLE DB Provider for DB2 does not support mapping DB2 bit strings and graphic data types to binary data types. The OLE DB Provider for DB2 does not support a binary-to-binary conversion. Consequently, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, and LONGVARCHAR FOR BIT DATA can only be supported when the Process Binary As Character property is set to true

when configuring the OLE DB data source or passed as part of the connection string. In this way, the OLE DB client would actually bind the column as an OLE DB character type, not as a binary type.

Using the OLE DB Provider for DB2, certain conversions of strings from EBCDIC to ASCII and then back to EBCDIC are asymmetric, and can result in strings that are different from the original. The EBCDIC specification contains ordinals for which there is no defined character. The OLE DB Provider for DB2 translates all such undefined characters to the question mark character ("?"). So when ASCII strings containing these characters are converted back to EBCDIC, these undefined characters will be replaced with question marks. To protect EBCDIC strings containing undefined characters, these fields should be tagged as binary strings and mapped by the application.

The ANSI to EBCDIC character conversions affected include the following:

Character Value (Decimal)	**Character Value (Hexadecimal)**	**ANSI Code Page 1252**	**EBCDIC Character After Conversion to CCSID 37**
128	0x80	Not used	?
130	0x82	Single low quote	?
131	0x83	Latin F with hook	?
132	0x84	Double low quote	?
133	0x85	Ellipsis	?
134	0x86	Dagger	?
135	0x87	Double dagger	?
136	0x88	Per mile	?
137	0x89	S with caron	?
138	0x8A	Left angle	?
139	0x8B	Ligature OE	?
140	0x8C	Not used	?
142	0x8E	Not used	?
145-156	0x91-0x9C		?
158-159	0x9E-0x9F		?

Floating Point Considerations Using the OLE DB Provider for DB2

When real or double (synonymous with float) data is inserted into a DB2 table as a floating point data type, it is stored in scientific notation. For example, FLOAT(1.1) would be stored as +1.10000E+000.

Care must be taken when executing SQL statements to make sure that the proper data type specified in the SQL statement matches the values stored in DB2. For example, the following select statement would match values in DB2 stored as decimal 1.1.

```
SELECT * FROM TEST WHERE C1 = 1.1
```

If the data in DB2 was stored as real numbers, there would not be a match since decimal 1.1 is stored as 1.1, not the representation of +1.10000E+000. When DB2 parses and executes the SQL select statement, it interprets 1.1 as a decimal type. When doing the select query, DB2 does not implicitly do the conversion to floating point. In this case, the SQL statement should explicitly typecast the 1.1 so that DB2 looks for the correct format (the scientific notation format). The select query would look like the following:

```
SELECT * FROM TEST WHERE C1 = REAL(1.1)
```

This will give the results expected. The SQL REAL function will convert the decimal 1.1 to the proper format before DB2 executes the actual select.

Usernames and Passwords Using the OLE DB Provider for DB2

When connecting to remote DB2 systems, most users must be authenticated by the remote system by passing a valid User ID and Password.

The AS/400 computer is case-sensitive with regard to User ID and Password. The AS/400 only accepts a User ID and Password in uppercase. The Microsoft OLE DB Provider for DB2 will force the User ID and Password into uppercase when it knows that it is connecting to a DB2/400 system.

The mainframe is not case-sensitive. This means that on mainframe computers, one can enter the User ID and Password in any case.

DB2 Universal Database (UDB) for Windows NT is case-sensitive. UDB supports mixed case Passwords. The user must enter the Password in the correct mixed case. When entering a User ID, use only the Windows NT user name and do not include the Windows NT Domain Name.

Troubleshooting the OLE DB Provider for DB2

The Windows 2000 and Windows NT Event Viewer can be a useful tool for troubleshooting data access in some cases. The OLE DB Provider for DB2 does not issue events. However, when SNA (APPC/LU 6.2) is used for the network transport for the OLE DB Provider for DB2, the low-level SNA APPC transport issues events on the SNA connection.

The Microsoft OLE DB Provider for DB2 supplied with Host Integration Server 2000 has the ability to trace DRDA data flows when used over TCP/IP.

This DB2 tracing capability is accessible from the SNADB2 Service tracing inside the Trace utility. This facility will show the same data as an APPC trace but without the control indicators (For example, What_Received). Socket errors are traced and the error codes can be looked up in Winsock2.h supplied with the Platform SDK.

The OLE DB Provider for DB2 can return the following types of errors:

- DB2 SQL errors from the remote database
- Microsoft OLE DB Provider-specific errors
- Errors from the underlying DRDA Application Requester network client

When the OLE DB Provider for DB2 passes an error code, the best source in which to look-up the meaning of the return code is often the SQL Reference or SQL Messages and Codes Reference for the target SQL database. In this case, the target database would be one of the DB2 platforms supported by the Microsoft OLE DB Provider for DB2.

The OLE DB Provider for DB2 maintains an internal integer variable named SQLCODE and an internal 5-byte character string variable named SQLSTATE used to check the execution of SQL statements on DB2. SQLCODE is set by DB2 after each SQL statement is executed. DB2 returns the following values for SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.
- SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.

SQLSTATE is also set by DB2 after the execution of each SQL statement. Application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE provides application programs with common codes for common error conditions (the values of SQLSTATE are product-specific only if the error or warning is product-specific). Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors.

SQLSTATE values consist of a two-character class code value, followed by a three-character subclass code value. The first character of an SQLSTATE value indicates whether the SQL statement was executed successfully or unsuccessfully (equal to or not equal to zero, respectively). Class code values represent classes of successful and unsuccessful execution conditions. The following SQLSTATE class codes are used by DB2:

Class Code	Description of Error Class
00	Successful completion. Execution of the SQL statement was successful and did not result in any type of warning or exception condition.
01	Warning
02	No data
07	Dynamic SQL error
08	Connection exception
0A	Feature not supported
0F	Invalid token
21	Cardinality violation
22	Data exception
23	Constraint violation
24	Invalid cursor state
25	Invalid Transaction State
26	Invalid SQL statement identifier
2D	Invalid transaction termination
34	Invalid cursor name

39	External function call exception
40	Transaction rollback
42	Syntax error or access rule violation
44	WITH CHECK OPTION violation
51	Invalid application state
53	Invalid operand or inconsistent specification
54	SQL or product limit exceeded
55	Object not in prerequisite state
56	Miscellaneous SQL or product error
57	Resource not available or operator intervention
58	System error

The SQLSTATE value of HY000 is defined as a provider-specific error. An SQLSTATE of 08S01 (connection exception with a subclass code of S01) also indicates a provider-specific error. This means the SQLCODE should be looked up in the driver-specific documentation included with the OLE DB Provider for DB2.

If the SQLSTATE does not indicate a driver-specific error when the OLE DB Provider for DB2 passes back an SQLSTATE of 08S01, it indicates a network error. For example, an SQLCODE of -603 is a provider-specific error that is mapped to DB2OLEDB_COMM_HOST_CONNECT_FAILED in the db2oledb.h include file supplied with the OLE DB Provider for DB2. Errors with an SQLSTATE of 08S01 are documented in the db2oledb.h include file (the SQLCODE value) which is located on the Host Integration Server 2000 CD-ROM in the SDK\Include subdirectory.

The following steps are useful in researching an error. Start by reading the provided error text returned by the OLE DB Provider for DB2. In some cases, the error text provides very limited useful information. For example, error text from an SQLCODE of -603 states the following:

```
Test connection failed because of an error in initializing provider.
Could not connect to specified host.
```

The next step is to lookup the SQLSTATE to determine the source of the error. Is the error a DB2 error, a network client error, or an OLE DB Provider error? An SQLSTATE of 08S01 is defined as follows:

```
Communication link failure.
```

This definition is intended to inform the user, administrator, or developer that the error is one related to the OLE DB Provider's underlying network client.

Unfortunately, many of the SQLSTATE codes returned by the OLE DB Provider for DB2 are DB2 errors and are not documented in the OLE DB Provider for DB2 on-line help.

The SQLSTATE of HY000 is defined as a provider-specific error. An SQLSTATE of 08S01 also indicates a provider-specific error. This means the SQLCODE should be looked up in the provider-specific documentation included with the OLE DB Provider for DB2.

If the SQLSTATE does not indicate a driver-specific error, then the SQLCODE should be looked up in the appropriate DB2 manual for the target platform. For example, an SQLCODE of -603 is documented in Appendix B, SQLCODEs and SQLSTATEs, in the *AS/400 Advanced Series DB2 for AS/400 SQL Programming, Version 4*, Document Number SC41-5611-00 published by IBM. An SQLCODE of -603 corresponds to SQLSTATE 23515 in the DB2 for OS/400 error code list. For example, the explanation for this SQLCODE is as follows:

```
Unique index cannot be created because of duplicate keys.
```

When the SQLSTATE and the SQLCODE definitions documented in these appendices create a mismatch with the actual errors returned, this usually indicates a provider-specific error condition.

A final step to understand an error is to check the db2oledb.h file. This file is not installed by the Host Integration Server or Host Integration Client setup program, but can be found on the Host Integration Server 2000 product CD ROM in the SDK\Include subdirectory. An SQLCODE (for example, -603) can be looked up by searching the right-most column of the db2oledb.h file for a value near to 603. In this case, one will see a comment "/* -600 */" and can then count down three additional lines to line number 603. The internal error code -603 is defined as follows:

```
DB2OLEDB_COMM_HOST_CONNECT_FAILED
```

Unfortunately, this error text is not further defined anywhere in the software or documentation provided to the customer. This particular error usually indicates a problem with the configuration parameters or the connection string passed.

ODBC Drivers

The Microsoft® ODBC Driver for DB2 enables users to access IBM Data Base 2 (DB2) from within an ODBC-aware application. ODBC defines a standard set of interfaces that provide access to disparate databases. The ODBC Driver for DB2 combines the data access of ODBC with the underlying Microsoft Distributed Relational Database Architecture (DRDA) Application Requester also used by the Microsoft OLE DB Provider for DB2. Using this combination of technologies, the ODBC Driver for DB2 can provide database access to IBM's Distributed Relational Database Architecture and IBM DB2.

Organizations have invested in secure, robust, enterprise-wide data storage and management systems. DRDA is a set of rules for distributing or extending relational data from one computer to another, such as a server computer to an IBM DB2 database server running on a mainframe or an AS/400 computer. By combining the ODBC and DRDA architectures, Microsoft allows organizations to preserve their investments in an existing data management infrastructure, while extending data access to all enterprise-wide DB2 data sources.

The ODBC Driver for DB2 can be used interactively or from an application program to issue SQL statements and execute DB2 stored procedures. From Microsoft Excel, users can import DB2 tables into worksheets and use Excel graphing tools to analyze the data. From Microsoft Access, users can import from and export to DB2. With Microsoft Internet Information Server (IIS), developers can publish DB2-stored information to users through a Web browser.

This section contains:

Goals of the ODBC Driver for DB2

Relational database management systems (RDBMS) are one of the major sources of mission-critical information in today's enterprise organizations. Relational database technology enables departments and individual users to save their information in centrally-managed database stores that can be easily maintained by the organization's information systems group.

IBM DB2 is a popular RDBMS for a significant number of enterprise customers. Customers need a cost-effective and manageable means to integrate DB2 with Microsoft SQL Server, Microsoft Internet Information Server (IIS), and Microsoft Office applications. The goal of Microsoft ODBC Driver for DB2 is to provide customers and solution providers with the means to integrate desktop database applications with this wealth of data residing on IBM DB2 database systems.

ODBC Driver for DB2 Architecture

The Microsoft® ODBC Driver for DB2 is an ODBC-compliant database driver for Windows 95, Windows 98, and Windows NT that lets your existing ODBC applications access data residing in IBM DB2 database servers without changing any code. The ODBC Driver for DB2 can connect ODBC-compliant applications with DB2 data sources using the underlying Microsoft Distributed Relational Database Architecture (DRDA) Application Requester. The ODBC application connects to the ODBC Driver for DB2. These ODBC requests are processed by the underlying Microsoft DRDA Application Requester. The data is then passed by an SQL interface to the DB2 data store.

The ODBC Driver for DB2 shares the same DRDA Application Requester that is used by the Microsoft OLE DB Provider for DB2. The DRDA Application Requester is the network client that provides remote database access to DB2 across an SNA LU6.2 and TCP/IP network.

The ODBC Driver for DB2 is compliant with the Microsoft Open Database Connectivity (ODBC) specification. ODBC is a specification for an application program interface (API) that enables applications to access multiple database systems using SQL.

Platforms Supported by the ODBC Driver for DB2

The Microsoft® ODBC Driver for DB2 supports popular DB2 platforms supported by the Microsoft OLE DB Provider for DB2 because both use the same underlying DRDA Application Requester (see Platforms Supported by the OLE DB Provider for DB2).

The ODBC Driver for DB2 offers network connectivity using SNA APPC LU6.2 connectivity, as well as native TCP/IP (not reliant on any special IBM or third-party routers).

ODBC Driver for DB2 Requirements

The ODBC Driver for DB2 supplied with Host Integration Server 2000 supports the following operating systems:

- Microsoft Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Professional
- Microsoft Windows NT® Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 6a or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later
- Microsoft Windows 98, Second Edition

The ODBC Driver for DB2 supplied with Host Integration Server 2000 Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

The ODBC Driver for DB2 supplied with Host Integration Server 2000 supports only the Windows 2000 and Windows NT (Intel version) platforms. Older versions of the ODBC Driver for DB2 that shipped with SNA Server 4.0 Service Pack 1 and later supported Windows NT on the Alpha architecture.

The ODBC Driver for DB2 requires the following PC-to-host connectivity software when connecting over SNA using LU 6.2:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server End-User Client
- Microsoft Host Integration Server Administrator Client

Microsoft Host Integration Server 2000 can be installed on Windows 2000 Server, Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows NT 4.0 Server, Windows NT 4.0 Server Enterprise Edition, or Windows NT 4.0 Server Terminal Server Edition.

The Microsoft Host Integration Server Administrator Client can be installed on Windows 2000 Professional or Windows NT 4.0 Workstation. The Microsoft Host Integration Server Administrator Client with Service Pack 1 can also be installed on Windows XP Professional. The Administrator Client cannot be installed on Windows 98 or Windows 95.

The Microsoft Host Integration Server End-User Client can be installed on Windows 2000 Professional, Windows NT 4.0 Workstation, or Windows 98. The Microsoft Host Integration Server End-User Client with Service Pack 1 can also be installed on Windows XP Professional, Windows XP Home Edition, or Windows Millennium Edition.

Note that the ODBC Driver for DB2 does not require any special host connectivity software when connecting directly to a host system using TCP/IP.

The ODBC Driver for DB2 supplied with Microsoft Host Integration Server 2000 supports the following ADO versions:

- ADO version 2.5. The Microsoft Host Integration Server 2000 data access features require the runtime libraries for ADO version 2.5. These libraries must be installed prior to installing the ODBC Driver for DB2. On Windows 2000, these ADO libraries are installed as part of the Windows 2000 operating system. On Windows NT 4.0 and Windows 98, these library files must be installed by running the Microsoft Data Access Components (MDAC) version 2.5 runtime package available as downloadable software from the Microsoft Universal Data Access Web site at http://go.microsoft.com/fwlink/?LinkId=12749.
  A version of the ADO 2.5 SDK is included in the Microsoft Data Access SDK which is available as a part of the Windows 2000 Platform SDK. These downloadable SDKs are available from the Microsoft Windows 2000 Web site at http://go.microsoft.com/fwlink/?LinkId=12752.

Configuring ODBC Data Sources

A data source associates a particular ODBC driver with the data to be accessed through that driver. Data source information must be configured for each DB2 system that is to be accessed using the ODBC Driver for DB2. The default parameters for the ODBC Driver for DB2 are used for the data source only when these parameters are not configured for each data source.

An ODBC data source name (DSN) can be one of the following types:

- User - A data source local to a computer and accessible only by the current user that created the data source.
- System - A data source local to a computer but not dedicated to a specific user, so any user with appropriate privileges can access a system DSN. A System data source is visible to all users on a computer, including Windows NT services.
- File - A data source stored in a file that can be shared among all users who have the same ODBC drivers installed. These data sources need not be dedicated to a specific user or local to a computer.

User and System data sources are stored in the registry. File data sources are stored as files with a file extension of dsn. File DSNs can be stored in any location on the file system including remotely-mounted shares. By default, File DSNs are stored in the following location:

C:\Program Files\Common Files\ODBC\Data Sources

ODBC data sources can be configured using the **ODBC Data Source Administrator**. On Windows 2000, a shortcut to the **ODBC Data Source Administrator** is located in the Control Panel under **Administrative Tools** as **Data Sources (ODBC)**. On Windows NT 4.0, a shortcut to the **ODBC Data Source Administrator** is located in the Control Panel as **Data Sources (ODBC)**. On Windows 98, a shortcut to the **ODBC Data Source Administrator** is located in the Control Panel as **ODBC Data Sources (32bit)**.

The NewSnaDS.exe utility provided as part of the ODBC Driver for DB2 enables users to create and modify ODBC data sources. This tool makes calls to the ODBC Data Source Administrator application to provide these functions.

The NewSnaDS tool is installed in the system folder below the subdirectory where Microsoft Host Integration Server 2000 is installed. The default location where this tool is installed is the following:

C:\Program Files\Host Integration Server\system\NewSnaDS.exe

A shortcut for using this tool to create or modify ODBC data sources is added to the **Programs** menu under **Host Integration Server\Data Integration** with a name of **ODBC Data Sources**. This shortcut is created when the Microsoft Host Integration Server 2000 or the Host Integration Client 2000 are first installed and support for data access is selected.

Several options are available for creating new data sources or modifying existing data sources for use with the ODBC Driver for DB2:

- Select the **ODBC Data Sources** shortcut under the **Programs\Host Integration Server\Data Integration** menu to start the NewSnaDS tool.
- Run the ODBC Data Source Administrator directly from the Control Panel.

The following steps describe how to configure a data source for use with the ODBC Driver for DB2 using the NewSnaDS tool:

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to Data Integration, and then click ODBC Data Sources to run the NewSnaDS tool.
3. Select the tab for the ODBC data source type to be created or modified: User DSN, System DSN, or File DSN.
4. If you are modifying an existing data source for the ODBC Driver for DB2, select the data source from the list box and click the Configure button.
   If you are creating a new data source, click the Add button, select the Microsoft ODBC Driver for DB2 from the list of drivers, and click the Finish button.
5. Data source information specific to the ODBC Driver for DB2 is configured using the Microsoft ODBC Driver for DB2 Configuration dialogs. Type the appropriate information into the various fields in the tabbed dialog box and clicking the OK button to return to the main ODBC Data Source Administrator application.
6. Configure other ODBC parameters not specific to the ODBC Driver for DB2 in the ODBC Data Source Administrator and click the OK button to save the data source.

The **Microsoft ODBC Driver for DB2 Configuration** dialog box contains five tabs, which are described in the following topics.

This section contains:

- General
- Connection
- Security
- Target Database
- Locale
- Configuration Property Mappings Between the ODBC Driver for DB2 and the OLE DB Provider for DB2

General

The **General** tab allows the user to configure the data source name required to connect to DB2. For the Microsoft ODBC Driver for DB2 supplied with Host Integration Server 2000, the **General** tab contains the following fields:

Parameter	Comments
Data Source Name	A blank field for specifying the name of the data source. Enter a string that identifies this ODBC data source. The data source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file, which is stored in C:\Program Files\Common Files\ODBC\Data Sources.
Description	A blank field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.

Connection

The **Connection** tab allows the user to configure the basic attributes required to connect to a data source. For the Microsoft ODBC Driver for DB2, the **Connection** tab has the following fields:

Parameter	Comments
Network transport	An option button (radio button) is used to select the network transport. Valid options are APPC Connection (SNA LU 6.2) or TCP/IP Connection. For the default, APPC Connection, the values for APPC local LU alias, APPC remote LU alias, and APPC Mode Name are required. For TCP/IP Connection, the values for IP address and Network port are required.
APPC local LU alias	When APPC Connection is selected, this field is the name of the local LU alias configured in Host Integration Server.
APPC remote LU alias	When APPC Connection is selected, this field is the name of the remote LU alias configured in Host Integration Server.
APPC mode name	When APPC Connection is selected, this field is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). The default is typically QPCSUPP.
IP address	When TCP/IP Connection is selected as the network transport, this field indicates the IP address of the host DB2 server.
Network port	When TCP/IP Connection is selected as the network transport, this field indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default is IP port 446.

The **Connection** tab also includes a **Test connection** button that may be used to test the connection parameters. A connection can only be tested after all of the required parameters for the **Connection** tab and other ODBC data source parameters are configured properly. When this button is clicked, a session is established with the remote DB2 system using ODBC Driver for DB2

Security

The **Security** tab allows the user to configure optional attributes used to restrict connections to a data source.

For the Microsoft ODBC Driver for DB2 in Host Integration Server 2000, the Security tab has the following fields:

Parameter	Comments
Authentication	An option button (radio button) is used to select the type of authentication. Valid options are **Use this username** or **Use single signon**. For the default Use this username option, the value for the username is required.
Use this username	When this option is selected, authentication is based on the username entered in the textbox. A valid user name is normally required to access data on DB2. A user name can remain optionally in the DSN. The ODBC Driver for DB2 will prompt the user at run time to enter a valid password. Additionally, the prompt dialog will enable the user to override the user name that is stored in the DSN.
Use single signon	An option button to select whether single sign-on or a specific user name should be used. Single sign-on is an optional Host Security feature. Single sign on enables the administrator to create data source definitions that isolate the logon process from the end user. The user context for single sign on is the user context associated with the SNA DB2 Service. When running on Windows 95 or Windows 98, the user context is associated with the currently logged-on user.

For the ODBC Driver for DB2 in SNA Server 4.0, the **Security** tab has the following fields:

Parameter	Comments
Use single sign on	An option button to select whether single sign-on or a specific user name should be used. Single sign-on is an optional SNA Server Host Security feature. Single sign on enables the administrator to create data source definitions that isolate the logon process from the end user. The user context for single sign on is the user context associated with the SNA DB2 Service. When running on Windows 95 or Windows 98, the user context is associated with the currently logged-on user.
User Name	The user name to use if the **Use single sign-on** option button is not selected. A valid user name is normally required to access data on DB2. A user name can remain optionally in the DSN. The ODBC Driver for DB2 will prompt the user at run time to enter a valid password. Additionally, the prompt dialog will enable the user to override the user name that is stored in the DSN.
Database is Read-only	This option creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example).

The AS/400 computer is case sensitive with regard to user IDs and passwords. When connecting to DB2 for OS/400, user names and passwords must be in uppercase. The AS/400 only accepts a DB2 for OS/400 user ID and password in uppercase. If a DB2 for OS/400 connection fails due to incorrect authentication, the ODBC driver resends the authentication, forcing the user ID and password into uppercase.

When connecting to DB2 on IBM mainframes, user names and passwords can be of mixed case; the mainframe is not case sensitive. The ODBC driver sends these values in uppercase.

DB2 Universal Database (UDB) for Windows NT is case sensitive. The user ID is stored in uppercase. The password is stored in mixed case and users must enter the password in the correct case. The ODBC driver sends the password exactly in the case entered by the user. The user ID should contain only the user name, not a combination of the Windows NT domain name and user name.

It is possible to connect using a specific User name and Password defined in DB2 on the host system or use the single sign-on feature (often referred to as integrated Windows security). If a specific DB2 User name and Password is to be used, this information may need to be saved to a data source name (DSN) file. The User name and Password are saved in clear text in the DSN file or to registry keys if a System or User DSN is selected. For security reasons when using file DSNs, it is imperative that the DSN file be protected with an Access Control List (ACL) that restricts access to only authorized users. System and User DSNs are preferred for security reasons as long as the locations where these ODBC DSNs are stored in the registry have appropriate

security protections. Saving the User name and Password in the DSN also forces this DSN to be updated whenever the Password associated with the User name is changed. So for a variety of reasons, specifying a User name and Password is not the preferred authentication option. Using the Single sign-on option is the preferred method for authentication.

Target Database

The **Target Database** tab allows the user to configure required, as well as optional, attributes used to define the target DB2 system.

For the Microsoft ODBC Driver for DB2 in Host Integration Server 2000, the Target Database tab has the following fields:

Parameter	Comments
Initial catalog	This parameter is used as the first part of a three-part fully qualified DB2 table name. It is referred to by different names depending on the DB2 platform. In DB2 for OS/390 and DB2 for MVS, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 that you need to connect to on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
Package collection	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This can be the same as the default schema. The ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver creates packages dynamically in the location that the user points to using the Package Collection parameter. By default, the ODBC Driver for DB2 automatically creates one package in the target collection, if one does not exist, at the time the user issues their first SQL statement. The package is created with GRANT EXECUTE authority to a single <AUTH_ID> only, where AUTH_ID is based on the user ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level based on the Isolation Level value specified in the connection. Problems can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (the PUBLIC group on the DB2 system, for example), then the package is created only for use by this user. This means that other users may be unable to access the required package. The solution is for an administrative user with package administrative rights to create a set of packages for use by all users (see Creating Packages for Use with the ODBC Driver for DB2). The ODBC Driver for DB2 supplied with Host Integration Server 2000 includes a tool program for use by administrators to create packages. The crtpkg.exe tool is a Windows GUI application for use by the administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. This utility will create a set of packages and grant EXECUTE privilege on these packages to the PUBLIC group representing all users on the DB2 system. The packages (see descriptions under the SQL_ATTR_TXN_ISOLATION connection attribute) created are as follows: AUTOCOMMITTED package (MSNC001 is only applicable on DB2/400) READ UNCOMMITTED package (MSUR001) READ COMMITTED package, (MSCS001) REPEATABLE READ package, (MSRS001) SERIALIZABLE package (MSRR001) Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400. Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE, the DB2 for OS/400 QSYS2.SYSPACKAGE, and the DB2 Universal Database (UDB) SYSIBM.SYSPACKAGE catalog tables. Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000. The package names changed from SNA Server 4.0.

Default schema	The name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses the Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (for example, ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at log on. For DB2/400, the driver uses QSYS2 if no collection is found matching the USER_ID value. This default is inappropriate in many cases so it is essential that the Default Schema value in the data source be defined.
Alternate TP Name	The Alternate Transaction Program (TP) Name property represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the Alternate TP Name in the off-line demo configuration (DRDADEMO.UDL). In that case, the Alternative TP Name is set to 0X07F9F9F9.
Distributed transactions	When this option is checked, two-phase commit (distributed unit of work) is enabled. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 V5R1 or later. This option also requires that the SNA LU 6.2 service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.
Process binary as character	When this option is checked, it indicates that binary data fields should be processed as characters. This option treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. See the **Locale** tab.

For the ODBC Driver for DB2 in SNA Server 4.0, the **Target Database** tab has the following fields:

Parameter	Comments
Remote Database Name	This parameter is used as the first part of a three-part fully qualified DB2 table name. It is referred to by different names depending on the DB2 platform. In DB2 for OS/390 and DB2 for MVS, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 that you need to connect to on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.

Package Collection	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This can be the same as the default schema. The ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver creates packages dynamically in the location that the user points to using the Package Collection parameter. By default, the ODBC Driver for DB2 automatically creates one package in the target collection, if one does not exist, at the time the user issues their first SQL statement. The package is created with GRANT EXECUTE authority to a single < AUTH_ID> only, where AUTH_ID is based on the user ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level based on the Isolation Level value configured in the data source. A problem can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (the PUBLIC group on the DB2 system, for example), then the package is created for use only by this user. This means that other users may be unable to access the required package. The solution is for an administrative user, with package administrative rights (for example., PACKADM authority in DB2 for OS/390), to create a set of packages for use by all users. The ODBC Driver for DB2 supplied with SNA Server 4.0 includes two utility programs for use by administrators to create packages. The crtpkg.exe tool is a command line utility for the administrator to create packages. The crtpkgw.exe tool is a Windows GUI utility used for the same purpose. Either of these utilities can be run using a privileged user ID to create packages in collections accessed by multiple users. These utilities will create a sets of packages (see descriptions under the Default Isolation parameter) and grant EXECUTE privilege on these packages to the PUBLIC group representing all users on the DB2 system. The packages created are as follows: AUTOCOMMIT package (SNANC001) READ_UNCOMMITTED package (SNACH001) REPEATABLE_READ package, (SNARR001) READ_COMMITTED package, (SNACS001) SERIALIZABLE or REPEATABLE_READ package (SNAAL001) Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE and DB2 for OS/400 QSYS.SYSPACKAGE.
Default Schema	The name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (for example, ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at log on. For DB2/400, the driver uses QSYS2 if no collection is found matching the USER_ID value. This default is inappropriate in many cases so it is essential that the Default Schema value in the data source be defined.

Default Isolation	This parameter determines the isolation level provided for the data source in cases of simultaneous access to DB2 objects by multiple applications. Valid settings for the default isolation level are: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2 for OS/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2 for OS/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2 for OS/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2 for OS/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Note that the ALL isolation level is not allowed. Users should set the isolation level to RS because it has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system).
Bind Type	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are: NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The NORM package binding option is designed to provide reasonable performance and maximum compatibility with different versions of DB2: DB2 for MVS, DB2 for OS/390, DB2 UDB, and DB2 for OS/400. Optionally, administrators can use the FAST method when running the Create Package utility and creating packages in many target collections. The FAST option should not be used with DB2 for MVS and DB2 UDB for Windows NT as a result of known incompatibilities. The default value for this parameter is NORM. This parameter is currently supported only by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2 or later and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
Alternate TP Name	The Alternate Transaction Program (TP) Name property represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
Auto commit	A check box indicating whether auto commit mode is enabled. This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true.
Convert all binary data types as character data types	This option treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. See the **Locale** tab.

Note that two-phase commit (distributed unit of work) and distributed transactions are not supported by the ODBC Driver for DB2 supplied with SNA Server 4.0.

Locale

The **Locale** tab allows the user to configure the parameters used for character conversion between the client and the DB2 server.

For the Microsoft ODBC Driver for DB2 in Host Integration Server 2000, the Locale tab has the following fields:

Parameter	Comments
Host CCSID	The coded character set identifier (CCSID) matching the DB2 data as represented on the remote computer. This property is required when processing binary data as character data. Unless the Process Binary as Character value is set, character data is converted based on the DB2 column CCSID and configured ANSI code page. This parameter defaults to U.S./Canada (37).
PC code page	This parameter indicates the personal computer code page to use. It is required when processing binary data as character data. Unless the Process Binary as Character value is set, character data is converted based on the default ANSI code page configured in Windows. The default value for this property is Latin 1 (1252).

Two versions of the Locale tab are possible for the ODBC Driver for DB2 supplied with SNA Server 4.0 depending the Service Pack installed.

For the ODBC Driver for DB2 in SNA Server 4.0 Service Pack 2, the Locale tab has the following fields:

Parameter	Comments
Host Locale	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the Process Binary as Character value is configured, character data is converted based on the DB2 column CCSID and default ANSI code page. This parameter defaults to U.S./Canada (37).
Use default code page for locale	A checkbox indicating whether the default ANSI code page for Windows should be used for the personal computer locale. If this default setting is not selected, then the user may choose any supported personal computer Code Page
PC Locale	When the previous check box is not selected, this parameter indicates which personal computer Code Page to use. This parameter is required when processing binary data as character data. Unless the Process Binary as Character value is configured, character data is converted based on the default ANSI code page configured in Windows. The default value for this property is Latin 1 (1252).

For the ODBC Driver for DB2 included with the Japanese version of SNA Server 4.0 Service Pack 2 and for the ODBC Driver for DB2 included with all versions of the SNA Server 4.0 Service Pack 3 or later, the **Locale** tab has the following fields:

Parameter	Comments
Host CCSID	The character code set identifier (CCSID) group box allows you to select the appropriate CCSID values matching the DB2 data as represented on the remote computer.
Single CCSID	The coded character set identifier (CCSID) matching the DB2 data as represented on the remote computer. This property is required when processing binary data as character data. Unless the Process Binary as Character value is set, character data is converted based on the DB2 column CCSID and configured ANSI code page. This parameter defaults to U.S./Canada (37).

Mixed CCSID	The mixed coded character set identifier (MCCSID) matching the DB2 data as represented on the remote computer. This property is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte character set (DBCS) data. When accessing DB2 for OS/390 or DB2 for MVS, a value must be specified for MCCSID if the "MIXED DATA" field (7) of the DB2 installation panel for Application Programming Defaults (DSNTIPF) is set to "YES." The following values for MCCSID are supported by the ODBC Driver for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is currently supported only by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2 or later and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
Graphics CCSID	The graphics coded character set identifier (GCCSID) matching the DB2 data represented on the remote computer. This property is required when accessing DB2 databases configured to support mixed single-byte and double-byte character set data. When accessing DB2 for OS/390 or DB2 for MVS, a value must be specified for MCCSID if the "MIXED DATA" field (7) of the DB2 installation panel for Application Programming Defaults (DSNTIPF) is set to "YES." The following values for GCCSID are supported by the ODBC Driver for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is currently supported only by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2 or later and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
PC Locale	This parameter indicates the personal computer Code Page to use. It is required when processing binary data as character data. Unless the Process Binary as Character value is set, character data is converted based on the default ANSI code page configured in Windows. The default value for this property is Latin 1 (1252).

Click the **OK** or **Cancel** button when data entry is finished. If you click the **OK** button, the values specified become the defaults when an application connects to this data source. These default values can be changed at any time using this procedure to reconfigure the data source. An ODBC application can override these defaults by connecting to the data source using a connection string with alternate values.

Configuration Property Mappings Between the ODBC Driver for DB2 and the OLE DB Provider for DB2

The following tables compare the configuration parameters used by the ODBC Driver for DB2 and the OLE DB Provider for DB2.

For the Microsoft ODBC Driver for DB2 in Host Integration Server 2000, the configuration parameters compare as follows:

Microsoft ODBC Driver for DB2	Microsoft OLE DB Provider for DB2
General	
Data Source Name	Data Source
Data Source Description	
Connection	
Connection	Network Transport Library
APPC Connection	SNA
TCP/IP Connection	TCPIP
APPC local LU Alias	APPC Local LU Alias
APPC remote LU Alias	APPC Remote LU Alias
APPC mode name	APPC Mode Name
IP address	Network Address
Network port	Network Port
Security	
Use this username	
User Name	User ID
Use Single Sign-on	Integrated Security.
Target Database	
Initial catalog	Initial Catalog
Package collection	Package Collection
Default schema	Default Schema
Alternate TP name	Alternate TP Name
Distributed transactions	Distributed transactions
Process binary as character	Process binary as character
Locale Tab	
Host CCSID	Host CCSID
PC code page	PC Code Page

For the Microsoft ODBC Driver for DB2 in SNA Server 4.0, the configuration parameters compare as follows:

Microsoft ODBC Driver for DB2	Microsoft OLE DB Provider for DB2
General	
Data Source Name	Data Source
Data Source Description	
Connection	
Connection	Network Transport Library
LU 6.2 Connection	SNA
TCP/IP Connection	TCPIP
Local APPC LU Alias	APPC Local LU Alias
Remote APPC LU Alias	APPC Remote LU Alias
APPC Mode Name	APPC Mode Name
IP Address	Network Address
Network Port	Network Port
Security	
Use this username	Persist Security Info
User Name	User ID
Password	Password

Use Single Sign-on	
Database is read-only	
Target Database	
Remote Database Name	Initial Catalog
Package Collection	Package Collection
Default Schema	Default Schema
Default Isolation	Default Isolation Level
Alternate TP Name	Alternate TP Name
BindType	BindType
Auto Commit	Auto Commit Mode
Convert Binary Data Types as Character Data Types	Process Binary as Character
Locale	
Host Locale	Host CCSID
Use default code page for locale	
PC Locale	PC Code Page
Locale (SNA Server 4.0 Service Pack 2 Japanese version and SNA Server 4.0 Service Pack 3 or later)	
System CCSID	Host CCSID
Mixed CCSID	Mixed CCSID
Graphics CCSID	Graphics CCSID
PC Locale	PC Code Page

ODBC Connection String Attributes

The ODBC **SQLBrowseConnect** and **SQLDriverConnect** functions allow passing in a connection string containing a series of attribute/value pairs to the ODBC Driver Manager to establish a connection with a data source. An example of a connection string :

```
"DSN=MYDATA;NTL=SNA;LLU=Local;RMU=Remote;RDB=BigData;PC=QSYS2;
DS=QSYS2;RO=false;UID=myname;PWD=Secret"
```

Some ODBC attributes are required as part of the connection string when used with the ODBC Driver for DB2.

The following tables compare the configuration parameters used by the ODBC Driver for DB2 and the ODBC attribute keywords that are supported by the OLE DB Driver for DB2 as part of the passed-in connection string.

For the ODBC Driver for DB2 in Host Integration Server 2000, these attribute keywords compare as follows:

Microsoft ODBC Driver for DB2	**ODBC Attribute Keyword**	**Comments**
General Tab		
Data Source Name	DSN	Required parameter.
Data Source Description	DESC	
Connection Tab		
Connection	NTL	Required parameter.
APPC connection	NTL=SNA	
TCP/IP connection	NTL=TCPIP	
APPC local LU alias	LLU	Applicable only if SNA (an APPC connection) is used for the network transport library (NTL=SNA).
APPC remote LU alias	RLU	Applicable only if SNA (an APPC connection) is used for the network transport library (NTL=SNA).
APPC mode name	MN	Applicable only if SNA (an APPC connection) is used for the network transport library (NTL=SNA).
IP address	NA	Applicable only if TCPIP (a TCP/IP connection) is used for the network transport library (NTL=TCPIP).
Network port	NP	Applicable only if TCPIP (a TCP/IP connection) is used for the network transport library (NTL=TCPIP).
Security Tab		
Use this username		
User Name	UID	
	PWD	The Password parameter is not on the Security Tab and is not configurable from the ODBC Administrator tool used to configure ODBC data sources. This parameter can only be preset using the ODBC connection string. Most applications will prompt the user for this parameter.
Use single sign-on		Not applicable
Target Database Tab		
Initial Catalog	RDB	Required parameter.
Package Collection	PC	Required parameter.

Default Schema	DS	Required parameter.
Alternate TP Name	TPN	
Distributed transactions	RUW	
Process binary as character	BAC	
Locale Tab		
Host CCSID	CCSID	Required parameter.
PC code page	CP	Required parameter.

For the ODBC Driver for DB2 supplied with Host Integration Server 2000, these attribute keywords compare as follows:

Microsoft ODBC Driver for DB2	**ODBC Attribute Keyword**	**Comments**
General Tab		
Data Source Name	DSN	Required parameter.
Data Source Description	DESC	
Connection Tab		
Connection	NTL	Required parameter.
LU 6.2 Connection	NTL=SNA	
TCP/IP Connection	NTL=TCPIP	
Local APPC LU Alias	LLU	Applicable only if SNA is used for the network transport library.
Remote APPC LU Alias	RLU	Applicable only if SNA is used for the network transport library.
APPC Mode Name	MN	Applicable only if SNA is used for the network transport library.
IP Address	NA	Applicable only if TCPIP is used for the network transport library.
Network Port	NP	Applicable only if TCPIP is used for the network transport library.
Security Tab		
Use this username		
User Name	UID	
Password	PWD	
Use Single Sign-on		Not applicable
Database is read only	RO	
Target Database Tab		
Remote Database Name	RDB	Required parameter.
Package Collection	PC	Required parameter.
Default Schema	DS	Required parameter.
Default Isolation	DIL	
Alternate TP Name	TPN	
BindType	BT	This parameter is only supported by the ODBC Driver included with SNA Server 4.0 Service Pack 2 Japanese version and SNA Server 4.0 Service Pack 3 or later.
Auto Commit	ACM	
Convert Binary Data Types as Character Data Types	BAC	
Locale Tab		
Host Locale	CCSID	Required parameter.
Use default code page for locale		
PC Locale	CP	Required parameter.
Locale (SNA Server 4.0 Service Pack 2 Japanese version and SNA Server 4.0 Service Pack 3 and later)		
System CCSID	CCSID	Required parameter.

Mixed CCSID	MCCSID	Required parameter.
Graphics CCSID	GCCSID	Required parameter.
PC Locale	CP	Required parameter.

Creating Packages for Use with the ODBC Driver for DB2

The ODBC Driver for DB2, which is implemented as an IBM Distributed Relational Database Architecture (DRDA) Application Requester, uses packages to issue SQL statements and call DB2 stored procedures. There is a configuration parameter that the ODBC Driver for DB2 uses to identify a location in which to create and store DB2 packages. The ODBC Driver for DB2 will create packages dynamically in the location to which the user points using the Package Collection parameter. This location may be configured using the Target Database tab from the Microsoft ODBC Data Source Administrator tool or can be passed as part of the ODBC connection string as an attribute keyword and argument. The attribute keyword for Package Collection is PC.

There are two package creation options:

1. The ODBC Driver for DB2 will auto-create one package for the currently-used isolation level at run-time if no package already exists. This auto-create process may fail if the user account does not have authority to create packages.
2. An administrator or user can manually creates all four packages (five packages on DB2/400) for use with all isolation levels and for use by all users (the PUBLIC group on DB2 representing all users) or a specific set of users. The ODBC Driver for DB2 includes a utility program for use by users with appropriate administrative privilege that will create these packages and grant access to the PUBLIC group for this purpose.

However, some users may not have the security level when manually creating packages to GRANT authority to the packages to other users (grant authority to the DB2 PUBLIC group representing all users, for example). This can be a problem if two or more users with different user IDs try to access a single collection of packages. The first user that created the packages will have access to the packages, but the second user likely will not. The Host Integration Server 2000 CD-ROM includes a program for use by an administrator or a user with appropriate privileges to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. The Create Packages for DB2 utility, CrtPkg, is a GUI-based tool included with Host Integration Server 2000 for creating packages for use with DB2. This tool (CrtPkg.exe) is installed in the System folder below the subdirectory where the Microsoft Host Integration Server 2000 has been installed. The default location where this tool is installed is the following:

Program Files\Host Integration Server\system\CrtPkg.exe

A shortcut for this tool is added to the **Programs** Menu off the Start button on the Windows Taskbar under the **Host Integration Server\Data Integration** folder with a name of **Packages for DB2**. This shortcut is created when the Microsoft Host Integration Server or the Host Integration Client are first installed and support for Data Access is checked.

This tool will create a set of packages and grant EXECUTE privileges on these packages to the PUBLIC group. The PUBLIC group on DB2 systems is a default group that represents all DB2 users. The following packages are created:

- AUTOCOMMITTED package (MSNC001) is only applicable on DB2/400)
- READ UNCOMMITTED package (MSUR001)
- READ COMMITTED package (MSCS001)
- REPEATABLE READ package (MSRS001)
- SERIALIZABLE package (MSRR001)

Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400.

The descriptive process name used by the CrtPkg utility of each package corresponds with the isolation levels defined in the ANSI SQL standard. The table below indicates how these packages correspond with the terms used by IBM for isolation levels in DB2 documentation.

Package Description	**Package Name**	**IBM Documentation**
AUTOCOMMITTED (Note that this applies only to DB2/400 and does not correspond with an ANSI SQL isolation level	MSNC001	COMMIT(*NONE) (NC). This isolation level is used in DB2/400 auto-commit mode only and has no corresponding isolation level on other DB2 platforms or in ANSI SQL.
READ UNCOMMITTED	MSUR001	UNCOMMITTED READ (UR). This isolation level corresponds with ANSI SQL READ UNCOMMITTED.

READ COMMITTED	MSCS001	CURSOR STABILITY (CS). This isolation level corresponds with ANSI SQL READ COMMITTED.
REPEATABLE READ	MSRS001	READ STABILITY (RS). This isolation level corresponds with ANSI SQL REPEATABLE READ.
SERIALIZABLE	MSRR001	REPEATABLE READ (RR). This isolation level corresponds with ANSI SQL SERIALIZABLE.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000. The package names used by the ODBC Driver for DB2 on SNA Server 4.0 are not compatible with the ODBC Driver for DB2 included with Host Integration Server. On SNA Server 4.0, these packages used different names as follows:

```
AUTOCOMMITTED package (SNANC001) only applicable on DB2/400
READ UNCOMMITTED package (SNACH001)
READ COMMITTED package, (SNACS001)
REPEATABLE READ package, (SNARR001)
SERIALIZABLE package (SNAAL001)
```

These Isolation Levels are described in detail under Support for Isolation Levels Using the ODBC Driver for DB2. These Isolation Levels are also described under the ADO IsolationLevel property. Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400.

Note that the CrtPkg tool creates this set of packages and grants EXECUTE privileges to PUBLIC. There may be cases for security reasons where EXECUTE privileges to this set of packages should be restricted to a certain group of users or specific users. In these cases, execution privileges on these created packages will need to be modified on the host system.

The CrtPkg utility will create all of these packages inside the Collection that is specified in the Package Collection property in the datalink file, or in the connection string. If the user does not have the appropriate authority to create packages in the specified Collection, or if the specified Collection does not exist, the ODBC Driver for DB2 will return an error.

In the case of DB2 on MVS or OS/390, the normal error text returned if the user does not the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your specific DB2 version for
a description of the associated Native Error and SQL State. SQLSTATE: 51002, SQLCODE: -567.
```

In the case of DB2/400, the normal error text returned if the user does not the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your specific DB2 version for
a description of the associated Native Error and SQL State. SQLSTATE: 51002, SQLCODE: -805.
```

In the case of DB2/400, the normal error returned if the collection does not exist would be as follows:

```
Failed to create AUTOCOMMITTED (NC) package.  RETCODE=-99.
SQL Error: Code=-204, State=42704, Error Text= A SQL error has occurred.  Please consult the
documentation for your specific DB2 version for a description of the associated Native Error
and SQL State. SQLSTATE: 42704, SQLCODE: -204
```

There are two authorities required to execute the create package process on OS/390 or MVS using the CrtPkg utility:

```
GRANT BINDADD TO <authorization ID>
GRANT CREATE IN COLLECTION <collection ID> TO <authorization ID>
```

The "authorization ID" is the user who needs the permission to create the packages. The "collection ID" is the name of the Collection, which the user specifies in the datalink file for the Package Collection property. This Collection should be a valid Collection within the DB2.
If an administrator executes the above statements on behalf a non-privileged user, then this non-privileged user can then run the CrtPkg utility. Once run, the CrtPkg process will create four sets of packages (one for each of the four isolation levels supported on DB2 for MVS or OS/390) for use by "all" (PUBLIC) users of the Microsoft data access features.

The example below illustrates this process on DB2 for MVS or DB2 for OS/390.

Grant rights to run the CrtPkg utility to authorization ID WNW999

```
GRANT BINDADD TO WNW999
GRANT CREATE IN COLLECTION MSPKG TO WNW999
```

Run the CrtPkg utility using authorization ID WNW999 (see the output from CrtPkg below)

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

In order to execute the CrtPkg utility on DB2/400, a user ID must have one of the following authorities:

- *CHANGE authority on the DB2 collection
- *ALL authority on the DB2 collection

If the user merely has *USE authority or if the user has *EXCLUDE authority, the Create Package process will fail.

There are several steps required to change user authority on a DB2/400 collection (AS/400 library): From interactive SQL (STRSQL command) while logged in as user with administrative privileges, create a new collection. This command can also be issued using ADO, OLE DB, and ODBC. However, most administrators typically create collections from the AS/400 console since the administrator must be logged in at the console to issue the Command Language (CL) command with which to change the user authority on the collection.

```
CREATE COLLECTION <collection ID>
```

From the AS/400 command console, issue the CL WRKOBJ command with the <collection ID> as a parameter.

```
WRKOBJ <collection ID>
```

The "collection ID" is the name of the Collection, which the user specifies in the datalink file for the Package Collection property. This Collection should be a valid Collection within DB2. The Work with objects screen appears. Place the cursor on the *PUBLIC Object Authority line and change the authority from *USE to *ALL.

If an administrator executes the above statements on behalf a non-privileged user, then this non-privileged user can then run the CrtPkg utility. Once run, the CrtPkg process will create five sets of packages (one for each of the five isolation levels supported on DB2/400) for use by "all" (PUBLIC) users of the Microsoft data access features. On DB2/400, five packages are created including the AUTOCOMMITTED packages.

The example below illustrates this process on DB2/400.

Grant rights to run the CrtPkg utility to authorization ID WNW999

```
CREATE COLLECTION MSPKG
WRKOBJ MSPKG
```

Run the CrtPkg utility (see the output from CrtPkg for DB2/400 below)

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating AUTOCOMMITTED (NC) package...
AUTOCOMMITTED (NC) package created.
Package creation succeeded.
EXECUTE privilege on MSNC001            granted to PUBLIC
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

CrtPkg allows a user to create a new DSN file or load a data source and modify an existing DSN file for connection configuration information. The File menu of CrtPkg has a New option used for creating a new ODBC DSN file or UDL File and a Load Data Source option to load an existing DSN or UDL file. The File menu Edit Data Source option allows a user to access and modify the parameters for a data source similar to using the NewSnaDS.exe tool. The Run menu option is used to create packages.

When using the create package tool, if the package collection specified does not exist, then DB2 returns SQLCODE -805.

When using auto-create packages, if a package collection is not specified or the package collection does not exist, then during the "auto-create" package process, the consumer application will receive SQLSTATE HY000 and SQLCODE -385. The SQLSTATE HY000 is defined as a driver-specific error. The -385 Error Return Code is not a SQLCODE but rather a DDM DRDA AR (DB2 client) return code. This error code is defined as DDM_VALNSPRM with the following associated text string:

```
"The parameter value is not supported by the target system."
```

The ODBC Driver for DB2 client error codes are defined in the db2oledb.h file located on the Host Integration Server 2000 CD-ROM.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000.

SNA Server 4.0 with Service Pack 3 came with two similar utilities for creating packages: CRTPKGW.EXE (a command-line tool) and CRTPKGW.EXE (a GUI-based tool).

ODBC Conformance

The Microsoft® ODBC Driver for DB2 supports ODBC 2.*x* and ODBC 3.*x* functions. SQL grammar conformance varies, depending on the version of the DB2 database that is accessed. The following sections list the ODBC functions and attributes supported by the Microsoft ODBC Driver for DB2.

This section contains:

Support for ODBC 2 Core Functions

The following table lists the ODBC 2.*x* Core functions that are supported by the Microsoft ODBC Driver for DB2.

ODBC 2.*x* Core Functions	**Functions Supported by the Microsoft ODBC Driver for DB2**
SQLAllocConnect	Yes
SQLAllocEnv	Yes
SQLAllocStmt	Yes
SQLBindCol	Yes
SQLCancel	Yes
SQLColAttributes	Yes
SQLConnect	Yes
SQLDescribeCol	Yes
SQLDisconnect	Yes
SQLError	Yes
SQLExecDirect	Yes
SQLExecute	Yes
SQLFetch	Yes
SQLFreeConnect	Yes
SQLFreeEnv	Yes
SQLFreeStmt	Yes
SQLGetCursorName	Yes
SQLNumResultCols	Yes
SQLPrepare	Yes
SQLRowCount	Yes
SQLSetCursorName	Yes
SQLSetParam	In ODBC 2.0, the ODBC 1.0 SQLSetparam function was replaced by SQLBindParameter.
SQLTransact	Yes

Support for ODBC 2 Level 1 Functions

The following table lists the ODBC 2.*x* level 1 functions that are supported by the Microsoft ODBC Driver for DB2.

ODBC 2.*x* Level 1 Functions	**Functions Supported by the Microsoft ODBC Driver for DB2**
SQLBindParameter	Yes
SQLColumns	Yes
SQLDriverConnect	Yes
SQLGetConnectOption	Yes
SQLGetData	Yes
SQLGetFunctions	Yes
SQLGetInfo	Yes
SQLGetStmtOption	Yes
SQLGetTypeInfo	Yes
SQLParamData	Yes
SQLPutData	Yes
SQLSetConnectOption	Yes
SQLSetStmtOption	Yes
SQLSpecialColumns	Yes
SQLStatistics	Yes
SQLTables	Yes

Support for ODBC 2 Level 2 Functions

The following table lists the ODBC 2.*x* level 2 functions that are supported by the Microsoft ODBC Driver for DB2.

ODBC 2.*x* Level 2 Functions Supported	**Functions Supported by the Microsoft ODBC Driver for DB2**
SQLBrowseConnect	No
SQLColumnPrivileges	No
SQLDataSources	Yes. This function is actually supported by the ODBC Driver Manager.
SQLDescribeParam	No
SQLDrivers	Yes. This function is actually supported by the ODBC Driver Manager.
SQLExtendedFetch	Yes, but supports forward scrolling only.
SQLForeignKeys	No
SQLMoreResults	Yes
SQLNativeSQL	Yes
SQLNumParams	Yes
SQLParamOptions	Yes
SQLPrimaryKeys	Yes, but SQL grammar conformance varies depending on the version of the DB2 database being accessed.
SQLProcedureColumns	No
SQLProcedures	Yes.
SetPos	Yes
SQLSetScrollOptions	Yes
SQLTablePrivileges	No

Support for ODBC 3 Functions

The following table lists the ODBC 3.0 functions that are supported by the Microsoft ODBC Driver for DB2.

ODBC 3.0 Functions	Functions Supported by the Microsoft ODBC Driver for DB2
SQLAllocHandle	Yes
SQLBulkOperations	No
SQLCloseCursor	Yes
SQLColAttribute	Yes
SQLCopyDesc	Yes
SQLEndTran	Yes
SQLFetchScroll	Yes, but supports forward scrolling only.
SQLFreeHandle	Yes
SQLGetConnectAttr	Yes
SQLGetDescField	Yes
SQLDescRec	Yes
SQLGetDiagField	Yes
SQLGetDiagRec	Yes
SQLGetEnvAttr	Yes
SQLGetStmtAttr	Yes
SQLRowCount	Yes.
SQLSetConnectAttr	Yes
SQLSetDescField	Yes
SQLSetDescRec	Yes
SQLSetEnvAttr	Yes
SQLSetStmtAttr	Yes

Support for ODBC Connection Attributes

The following table lists the ODBC connection attribute support using the Microsoft ODBC Driver for DB2. Note that the connection attributes in this list use the ODBC Version 3.0 attribute names, rather than the older ODBC 1.0 names.

ODBC Connection Attribute	ODBC Version	Attribute Supported by the Microsoft ODBC Driver for DB2
SQL_ATTR_ACCESS_MODE	1.0	Yes
SQL_ATTR_ASYNC_ENABLE	3.0	No
SQL_ATTR_AUTO_IPD	3.0	No
SQL_ATTR_AUTOCOMMIT	1.0	Yes
SQL_ATTR_CONNECTION_DEAD	3.5	No
SQL_ATTR_CONNECTION_TIMEOUT	3.0	No
SQL_ATTR_CURRENT_CATALOG	2.0	Yes
SQL_ATTR_ENLIST_IN_DTC	3.0	Yes
SQL_ATTR_ENLIST_IN_XA	3.0	No
SQL_ATTR_LOGIN_TIMEOUT	1.0	No
SQL_ATTR_METADATA_ID	3.0	No
SQL_ATTR_ODBC_CURSORS	2.0	Yes, handled by ODBC Driver Manager.
SQL_ATTR_PACKET_SIZE	2.0	No
SQL_ATTR_QUIET_MODE	2.0	Yes
SQL_ATTR_TRACE	1.0	Yes, handled by ODBC Driver Manager.
SQL_ATTR_TRACEFILE	1.0	Yes, handled by ODBC Driver Manager.
SQL_ATTR_TRANSLATE_LIB	1.0	No
SQL_ATTR_TRANSLATE_DLL	1.0	No
SQL_ATTR_TRANSLATE_OPTION	1.0	No
SQL_ATTR_TXN_ISOLATION	1.0	Yes

Support for ODBC Statement Attributes

The following table lists the ODBC statement attribute support using the Microsoft ODBC Driver for DB2. Note that the statement attributes in this list use the ODBC Version 3.0 attribute names, rather than the older ODBC 1.0 names.

ODBC Statement Attribute	ODBC Version	Attribute Supported by the Microsoft ODBC Driver for DB2
SQL_ATTR_APP_PARAM_DESC	3.0	Yes
SQL_ATTR_APP_ROW_DESC	3.0	Yes
SQL_ATTR_ASYNC_ENABLE	1.0	No
SQL_ATTR_CONCURRENCY		No
SQL_ATTR_CURSOR_SCROLLABLE	3.0	No
SQL_ATTR_CURSOR_SENSITIVITY	3.0	No
SQL_ATTR_CURSOR_TYPE	1.0	Yes, but the ODBC Driver for DB2 supports a forward only cursor type.
SQL_ATTR_ENABLE_AUTO_IPD	3.0	No
SQL_ATTR_FETCH_BOOKMARK_PTR	3.0	No
SQL_ATTR_IMP_PARAM_DESC		Yes, handled by ODBC Driver Manager.
SQL_ATTR_IMP_ROW_DESC		Yes, handled by ODBC Driver Manager.
SQL_ATTR_KEYSET_SIZE	1.0	No
SQL_ATTR_MAX_LENGTH	1.0	No
SQL_ATTR_MAX_ROWS	1.0	No
SQL_ATTR_METADATA_ID	3.0	Yes
SQL_ATTR_NOSCAN	1.0	Yes
SQL_ATTR_PARAM_BIND_OFFSET_PTR	3.0	Yes
SQL_ATTR_PARAM_BIND_TYPE	3.0	Yes
SQL_ATTR_PARAM_OPERATION_PTR	3.0	Yes
SQL_ATTR_PARAM_STATUS_PTR	3.0	Yes
SQL_ATTR_PARAMS_PROCESSED_PTR	3.0	Yes
SQL_ATTR_PARAMSET_SIZE	3.0	Yes
SQL_ATTR_QUERY_TIMEOUT	1.0	No
SQL_ATTR_RETRIEVE_DATA	1.0	No
SQL_ATTR_ROW_ARRAY_SIZE	3.0	Yes
SQL_ATTR_ROW_BIND_OFFSET_PTR	3.0	Yes
SQL_ATTR_ROW_BIND_TYPE	1.0	Yes
SQL_ATTR_ROW_NUMBER	1.0	No
SQL_ATTR_ROW_OPERATION_PTR	3.0	No
SQL_ATTR_ROW_STATUS_PTR	3.0	Yes
SQL_ATTR_ROWS_FETCHED_PTR	3.0	Yes
SQL_ATTR_SIMULATE_CURSOR	1.0	No
SQL_ATTR_USE_BOOKMARKS	1.0	No

Programming Considerations Using the ODBC Driver for DB2

The Microsoft ODBC Driver for DB2 provides pass through support for SQL statements. No SQL parsing is provided. The user must know what SQL syntax is supported for the target DB2 implementation. For information on what SQL syntax is supported, see the specific DB2 SQL Reference and DB2 Application Programming and SQL Guide for the DB2-specific platform.

The ODBC Driver for DB2 does not parse the SQL statements to qualify table names. Consequently, users of the ODBC Driver for DB2 must use either two-part or three-part (fully-qualified) object names when naming tables, views, and stored procedures in DB2. A two-part table name would consist of the user ID and table, <UserID>.<Table>. One-part names (just the table name) will not succeed unless the combination of the DB2 collection and schema name correspond directly to the ODBC User ID

The Microsoft ODBC Driver for DB2 does not insert the correct value for the fraction when using a parameterized insert with the TIMESTAMP data type.

The Microsoft Data Access Components (MDAC) support the option of using a client cursor engine. This service component is implemented as part of OLE DB, ADO, and Remote Data Services (RDS). When using ADO, a client cursor is enabled by setting the CursorLocation property on the recordset to adUseClient. When using the ADO Client Cursor Engine with DB2 for OS/390, the developer must configure the ODBC Driver for DB2 Auto Commit Mode property in the DSN or connection string to FALSE. This is not required when connecting to DB2 for OS/400.

The ODBC Driver for DB2 included with Host Integration Server 2000 supports updating capabilities when used with a client cursor engine and the following requirements are met:

- To support updates (UPDATE, INSERT, and DELETE) , the values in at least one column in the target table must be unique.
- The Auto Commit parameter must be set to FALSE when configuring the data source or when this parameter is passed as part of a connection string.

Previous versions of the ODBC Driver for DB2 do not support any updating capabilities when used with a client cursor engine. In other words, if a client cursor engine is enabled using RDS or ADO, the ODBC Driver for DB2 cannot be used to update data on the host. The ADO recordset is treated as if it were read-only. When using the ADO Client Cursor Engine with DB2 for OS/390, the developer must configure the ODBC Driver for DB2 Auto Commit parameter in the data source or connection string to FALSE. This is not required when connecting to DB2 for OS/400.

When the intent is to update records with a server-side cursor, DB2 requires that the SQL SELECT statement also include the FOR UPDATE option. For example, to select all records from the AUTHORS table in the DB2 collection called PUBS with an intent to update requires the following SQL syntax:

```
SELECT * FROM PUBS.AUTHORS FOR UPDATE
```

When using DB2 for MVS V4R1 and DB2 for OS/400 V3R2, there are further requirements to indicate the columns that you intend to update. For example, to update the AU_LNAME and AU_FNAME columns in the PUBS.AUTHORS table, the following SQL syntax must be used:

```
SELECT * FROM PUBS.AUTHORS FOR UPDATE OF AU_LNAME, AU_FNAME
```

The Microsoft ODBC Driver for DB2 provides support for distributed transactions and DRDA Distributed Unit of Work, and can participate in a distributed transaction coordinated by Microsoft Distributed Transaction Coordinator. This feature is only available when connecting to one of the following across an LU 6.2 network connection:

- DB2 for OS/390 V5R1 or later.
- DB2 for DB2/400 V4R3 or later.

This option also requires that the SNA LU 6.2 service is selected as the network transport and Microsoft Transaction Server (MTS) is installed. The Microsoft ODBC Driver for DB2 does not support automatic transaction enlistment under Microsoft Transaction Server.

Applications should not commit or roll back transactions by executing COMMIT or ROLLBACK statements using the SQLExecute or SQLExecDirect ODBC functions. The effects of doing this are undefined and the ODBC Driver for DB2 no longer knows when a transaction is active. Applications should call the SQLEndTran ODBC function instead.

Microsoft Visual Studio 6.0 offers a number of ADO data-bound controls, including a datagrid and the ADO Data Control. When

using these ADO data controls, the developer must set the CursorLocation property on the recordset to adUseClient. Additionally, when using these ADO data controls with DB2 for OS/390, the developer must set the ODBC Driver for DB2 Auto Commit parameter in the data source or connection string to FALSE.

The Microsoft ODBC Driver for DB2 is not supported when used in conjunction with the Microsoft SQL Server 7.0 Replication feature. In place of the ODBC driver, use the Microsoft OLE DB Provider for DB2.

Microsoft Query (MSQUERY) supplied with Excel and Office can be used to access ODBC data sources using the ODBC Driver for DB2. When a data of a column defined with TIMESTAMP data type is updated using Microsoft Query, the microseconds portion of the TIMESTAMP is not updated properly. The net result is that updating any TIMESTAMP value using Microsoft Query will cause the loss of fractional seconds.

The Microsoft ODBC Driver for DB2 is not supported for use with the Microsoft ODBC .NET Data Provider. When accessing DB2 from ADO.NET, use the Microsoft OLE DB Provider for DB2 in conjunction with the Microsoft OLE DB .NET Data Provider.

This section contains:

Stored Procedure Support Using the ODBC Driver for DB2

The Microsoft ODBC Driver for DB2 supports calling DB2 stored procedures. An application must use the CALL keyword before the SQL statement in order to execute a stored procedure. When using ADO, a CommandType property of adCmdStoredProc cannot be used for executing a stored procedure since ADO inserts an EXEC not CALL keyword before the command text. In order to execute a stored procedure using ADO, the **CommandType** property should be set to adCmdText and the CALL keyword should be used before the SQL statement containing the stored procedure to be executed.

When calling DB2 stored procedures using the ODBC Driver for DB2, the following limitations apply:

- Binding output parameters of types REAL or DOUBLE are not supported.
- Calling stored procedures when the parameter values contain CHAR Mixed or GRAPHIC (DBCS) data types are not supported.
- Calling a non-existent procedure causes error.
- The ODBC Driver for DB2 does not return single or multiple result sets.

Support for Isolation Levels Using the ODBC Driver for DB2

The Microsoft ODBC Driver for DB2 provides flexibility in dealing with issues of isolation levels and transaction state. The ODBC **SQLSetConnectAttr** function is used to set the isolation level that is to be used for a connection. This function would be called with the attribute parameter set to SQL_ATTR_TXN_ISOLATION and the ValuePtr parameter pointing to an integer value indicating the isolation level requested. This integer value is a 32-bit bitmask that sets the transaction isolation level for the current connection.

The allowable values for isolation level (the ValuePtr parameter when calling SQLSetConnectAttr) can be determined by calling SQLGetInfo with InfoType equal to SQL_TXN_ISOLATION_OPTION. The following table list the allowable values for isolation level using the ODBC Driver for DB2 supplied with Host Integration Server.

ODBC Isolation Level Attribute	**Description**
SQL_TXN_READ_COMMITTED	When this attribute value is set, it isolates any data read from changes by others and changes made by others by others cannot be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read. This is the default value for isolation level This isolation level is also called Cursor Stability (CS) in IBM DB2 documentation.
SQL_TXN_READ_UNCOMMITTED	When this attribute value is set, it does not isolate data read from changes by others and changes made by others by others can be seen. The re-execution of the read statement is affected by others. This does not support a repeatable read. This isolation level is called Uncommitted Read (UR) in IBM DB2 documentation.
SQL_TXN_REPEATABLE_READ	When this attribute value is set, it isolates any data read from changes by others and changes made by others cannot be seen. The re-execution of the read statement is affected by others. This supports a repeatable read. This isolation level is called Read Stability (RS) in IBM DB2 documentation.
SQL_TXN_SERIALIZABLE	When this attribute value is set, it isolates any data read from changes by others and changes made by others by others cannot be seen. The re-execution of the read statement is not affected by others. This supports a repeatable read. This isolation level is called Repeatable Read (RR) in IBM DB2 documentation.

The SQL_ATTR_TXN_ISOLATION attribute can be set only if there are no open transactions on the connection. An application must call **SQLEndTran** to commit or roll back all open transactions on a connection, before calling **SQLSetConnectAttr** with this option.

Some connection attributes support substitution of a similar value if the data source does not support the value specified in *ValuePtr*. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). To determine the substituted value, an application calls SQLGetConnectAttr.

Code Page Support Using the ODBC Driver for DB2

When creating data sources or file DSNs for use with the ODBC Driver for DB2, the Host character code set identifier (CCSID) should be configured in the data source to match the DB2 data as represented on the remote host computer. The Host CCSID parameter defaults to EBCDIC U.S./Canada (37).

Depending on the version of Windows being used, to support specific code page conversions, you may need to install the appropriate National Language Support (NLS) file for your locale.

On Windows 2000, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows 2000.

On Windows NT 4.0, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows NT or when you install the Windows NT Language Pack on a non-localized version of Windows NT. The Windows NT Language Pack is available on the Windows NT 4.0 CD-ROM in the LANGPACK directory. You install the locale components of the language pack as needed by either making a change in the Control Panel Locales applet or by installing one of the locale-specific INF files.

On Windows 98 and Windows 95, the appropriate ANSI NLS file for your locale is installed automatically when you install a localized version of Windows 98 or Windows 95.

The following sections discuss the character code set identifiers (CCSIDs) supported by ODBC Driver for DB2 in Host Integration Server 2000.The The tables in these sections list the INF files by name that are required under Windows NT 4.0 for a specific codepage (european.inf, for example). Typically you would install locales one at a time as needed.

This section contains:

ANSI Code Page Support Using the ODBC Driver for DB2

IBM DB2 Universal Database for Windows NT and IBM DB2 Universal Database for AIX are frequently configured to use ANSI code pages, for example ANSI 1253 (Greek). Host Integration Server 2000 includes support for some ANSI code pages for purposes of ANSI-to-UNICODE-to-ANSI conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These ANSI code pages can be used when accessing IBM DB2 Universal Database on Windows NT and IBM DB2 ON AIX (not all of these ANSI code pages are supported on IBM DB2 Universal Database for AIX).

The following table shows the ANSI character code set identifiers (CCSIDs) supported by ODBC Driver for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
ANSI - Arabic	1256	1256	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
ANSI - Baltic	1257	1257	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ANSI - Cyrillic	1251	1251	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
ANSI - Central Europe	1250	1250	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ANSI - Greek	1253	1253	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
ANSI - Hebrew	1255	1255	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
ANSI - Latin I	1252	1252	Support for this codepage is normally installed as part of the operating system on Windows 2000, Windows NT, Windows 98, and Windows 95. On Windows NT 4.0, support for this NLS Code Page is installed using the us_eng.inf file from the Language Pack.
ANSI - Turkish	1254	1254	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
ANSI/OEM - Japanese Shift JIS	932	932	On Windows NT 4.0, support for this NLS Code Page is installed using the japanese.inf file from the Language Pack.
ANSI/OEM - Korean	949	949	On Windows NT 4.0, support for this NLS Code Page is installed using the korean.inf file from the Language Pack.
ANSI/OEM - Simplified Chinese GBK	936	936	On Windows NT 4.0, support for this NLS Code Page is installed using the exchsrvr.inf file from the Language Pack.
ANSI/OEM - Thai	874	874	On Windows NT 4.0, support for this NLS Code Page is installed using the thai.inf file from the Language Pack.
ANSI/OEM - Traditional Chinese Big5	950	950	On Windows NT 4.0, support for this NLS Code Page is installed using the tchinese.inf file from the Language Pack.
ANSI/OEM - Viet Nam	1258	1258	On Windows NT 4.0, support for this NLS Code Page is installed using the vietnam.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows 2000 or Windows NT definitions for these NLS files. The Microsoft NLS Code Page column represents the code page number that is registered and associated with an ANSI-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the ODBC Driver for DB2. When setting the Host CCSID or PC Code Page parameter using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the ANSI code page in IBM publications, which for these supported ANSI CCSIDs are the same as the Microsoft CCSID values. IBM lists their ANSI support in publications by referencing the display name which for these ANSI code pages is the same as the Microsoft display name. The ODBC Driver for DB2 does not recognize or display the IBM CCSID values when configuring data sources. The ODBC Driver for DB2 maps the Microsoft NLS numbers to ANSI NLS files which correspond with the appropriate IBM CCSID numbers. The Microsoft ODBC Driver for DB2 passes the corresponding IBM CCSID to the DB2 system at run time even though you configure the driver to use the Microsoft NLS number.

These are the only ANSI pages currently supported by the ODBC Driver for DB2 in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. IBM supports additional ANSI pages, however, the ANSI code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM ANSI code pages (CCSIDs) match.

EBCDIC Code Page Support Using the ODBC Driver for DB2

IBM DB2 for MVS, IBM DB2 for OS/390, and IBM DB2 for OS/400 are frequently configured to use EBCDIC code pages, for example EBCDIC 875 (Greek Modern). Host Integration Server 2000 includes support for most EBCDIC code pages for purposes of EBCDIC-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-EBCDIC, and EBCDIC-to-UNICODE-to-EBCDIC conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These EBCDIC code pages can be used when accessing IBM DB2 on a variety of platforms (not all of these EBCDIC code pages are supported on all versions of IBM DB2).

The following table shows the EBCDIC character code set identifiers (CCSIDs) supported by ODBC Driver for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
IBM EBCDIC - Arabic	20420	420	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
IBM EBCDIC - Cyrillic (Russian)	20880	880	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
IBM EBCDIC - Cyrillic (Serbian, Bulgarian)	21025	1025	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
IBM EBCDIC - Denmark/Norway	20277	277	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Denmark/Norway (Euro)	1142	1142	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Finland/Sweden	20278	278	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Finland/Sweden (Euro)	1143	1143	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - France	20297	297	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - France (Euro)	1147	1147	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Germany	20273	273	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Germany (Euro)	1141	1141	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Greek	20423	423	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
IBM EBCDIC - Greek (Modern)	875	875	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
IBM EBCDIC - Hebrew	20424	424	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
IBM EBCDIC - Icelandic	20871	871	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Icelandic (Euro)	1149	1149	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - International	500	500	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - International (Euro)	1148	1148	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Italy	20280	280	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Italy (Euro)	1144	1144	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Japan English/Kanji (Extended)	939	939	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japan English/Kanji (Extended)	5035	5035	Support for this double-byte character set is supplied using TRNSDT.

IBM EBCDIC - Japan Katakana/Kanji (Extended)	930	930	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japan Katakana/Kanji (Extended)	5026	5026	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Japanese	931	931	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Korea (Extended)	933	933	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Latin America/Spain	20284	284	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Latin America/Spain (Euro)	1145	1145	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - Multilingual/ROECE (Latin-2)	870	870	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - Simplified Chinese (Extended)	935	935	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Thai	20838	838	On Windows NT 4.0, support for this NLS Code Page is installed using the thai.inf file from the Language Pack.
IBM EBCDIC - Traditional Chinese (Extended)	937	937	Support for this double-byte character set is supplied using TRNSDT.
IBM EBCDIC - Turkish (Latin-3)	20905	905	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
IBM EBCDIC - Turkish (Latin-5)	1026	1026	On Windows NT 4.0, support for this NLS Code Page is installed using the turkish.inf file from the Language Pack.
IBM EBCDIC - U.S./Canada	037	37	On Windows NT 4.0, support for this NLS Code Page is installed using the us_eng.inf file from the Language Pack.
IBM EBCDIC - U.S./Canada (Euro)	1140	1140	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.
IBM EBCDIC - United Kingdom	20285	285	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
IBM EBCDIC - United Kingdom (Euro)	1146	1146	On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows 2000 or Windows NT definitions for these NLS files. The Microsoft NLS Code Page column represents the code page number that is registered and associated with an EBCDIC-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the ODBC Driver for DB2. When setting the Host CCSID or PC Code Page parameter using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the EBCDIC code page in IBM publications. IBM lists their EBCDIC support in publications by referencing the display name which for these EBCDIC code pages is the same as the Microsoft display name. The ODBC Driver for DB2 does not recognize or display the IBM CCSID values when configuring data sources using data links. The ODBC Driver for DB2 maps the Microsoft NLS numbers to EBCDIC NLS files which correspond with the appropriate IBM CCSID numbers. The Microsoft ODBC Driver for DB2 passes the corresponding IBM CCSID to the DB2 system at run time even though you configure the driver to use the Microsoft NLS number.

These are the only EBCDIC pages currently supported by the ODBC Driver for DB2 in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. IBM supports additional EBCDIC pages, however, the EBCDIC code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM EBCDIC code pages (CCSIDs) match.

ISO Code Page Support Using the ODBC Driver for DB2

IBM DB2 Universal Database for Windows NT and IBM DB2 Universal Database for AIX are frequently configured for an ISO code page, for example ISO 819 (Latin I). Host Integration Server 2000 includes support for some ISO code pages for purposes of ISO-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-ISO, and ISO-to-UNICODE-to-ISO conversions when using the OLE DB Provider for DB2 or the ODBC Driver for DB2. These ISO code pages can be used when accessing IBM DB2 Universal

The following table shows the ISO character code set identifiers (CCSIDs) supported by ODBC Driver for DB2 in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
ISO 8859-1 Latin 1	28591	819	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-2 Central Europe	28592	912	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-5 Cyrillic	28595	915	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
ISO 8859-6 Arabic	28596	1089	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
ISO 8859-7 Greek	28597	813	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
ISO 8859-8 Hebrew	28598	916	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
ISO 8859-9 Turkish	28599	920	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 6937 Non-Spacing Accent	20269	819	Note that ISO 6937 (CCSID 20269) is not supported by the ODBC Driver for DB2, but is displayed in the list of configuration options when creating or modifying data sources.
ISO 8859-15 Latin 9 (Euro)	20865	923	NLS Code Page 819 with support for the Euro. On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows NT Language Pack definitions for these NLS files.

The Microsoft NLS Code Page column represents the code page number that is registered and associated with an ISO-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when configuring data sources when using the ODBC Driver for DB2. When setting the Host CCSID or PC Code Page attribute/property using a connection string, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the ISO code page in IBM publications. IBM lists their ISO support in publications by referencing the locale name (Bulgaria for ISO8859-5 and 915, for example) rather than simply using ISO 8859-5 Cyrillic as used by Microsoft. The ODBC Driver for DB2 does not recognize or display the IBM CCSID values when configuring data sources. The ODBC Driver for DB2 maps the Microsoft NLS numbers to ISO NLS files which correspond with the appropriate IBM CCSID numbers. The Microsoft ODBC Driver for DB2 passes the corresponding IBM CCSID to the DB2 system at run time even though you configure the driver to use the Microsoft NLS number.

Note that IBM CCSID 819 is associated with both ISO 8859-1 Latin 1 and ISO 6937 Non-Spacing Accent. It is up to the user to choose the standard ISO 8859-1 Latin 1 code page by selecting NLS code page 28591 or the modified code page ISO 6937 Non-Spacing Accent by selecting NLS code page 20269. Note that ISO 6937 Non-Spacing Accent (CCSID 20269) is not currently supported by the ODBC Driver for DB2, but is displayed in the configuration options when creating or modifying data sources.

IBM CCSID 916 (ISO 8859-8) supports Hebrew "visual sort order". IBM CCSID 920 (ISO 8859-8 derivation) supports Hebrew "logical sort order". Although Microsoft supports the logical sort order with NLS 38598, this NLS file is only distributed with Internet Explorer 5 or Windows 2000. The ODBC Driver for DB2 has not been tested using the ISO 8859-8 derivation matching IBM CCSID 920 and does not support this configuration.

These are the only ISO pages currently supported in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. Microsoft supports a number of additional ISO pages. IBM also supports additional ISO pages. However, the code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM CCSIDs match.

DBCS Code Page Support Using the ODBC Driver for DB2

Support for Double-Byte Character String (DBCS) data is limited using the ODBC Driver for DB2. Conversions between DBCS and ANSI code pages are not supported and conversions between DBCS and ISO code pages are not supported. Positioned updates against DBCS EBCDIC implementations of DB2 are not supported.

The DB2 GRAPHIC data types (GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC) are not supported. These DB2 data types support DBCS (not mixed) data. Mixed data types are supported using CHAR FOR MIXED DATA, VARCHAR FOR MIXED DATA, and LONGVARCHAR FOR MIXED DATA.

Parameterized SQL statements or calling stored procedures when the parameter values contain Mixed or DBCS characters are not supported.

Data Conversion Using the ODBC Driver for DB2

The design of ODBC APIs are similar to other ISAM APIs. The APIs are handle-based. After opening a file, the application can determine the buffer size required to store a row, use the cursor APIs to move, and optionally retrieve one or more rows of data using the row-level binding.

Data is converted to default SQL data types, as defined in ODBC, and listed in the following table:

DB2 data type	Default SQL Data Type	Exposed as Native Type in SQLGetTypeInfo	Comments
BIGINT			An eight-byte integer. This data type is not supported by the ODBC Driver for DB2.
BLOB			A Binary Large Object (BLOB) is a varying-length string that can be up to 2 gigabytes in length. A BLOB is primarily intended to hold binary data. This data type is not supported by the ODBC Driver for DB2.
CHAR (BIT)	SQL_BINARY	No	A fixed length (double-byte only) character string.
CHAR (SBCS)	SQL_CHAR	Yes	A fixed-length SBCS character string.
CHAR (MIXED)	SQL_CHAR	No	A fixed-length mixed (single and double-byte) character string.
CLOB			A Character Large Object (CLOB) is a varying-length string that can be up to 2 gigabytes in length. A CLOB is used to store large single-byte character set data. A CLOB is considered to be a character string. It is not supported by the ODBC Driver for DB2.
DATE	SQL_TYPE_DATE	Yes	A ten byte date string. This data type is converted to an SQL_DATE for use by ODBC.
DBCLOB			A Double-Byte Character Large Object (DBCLOB) is a varying-length string of double-byte characters that can be up to 2 gigabytes in length (1,073,741,823 double-byte characters). A DBCLOB is used to store large double-byte character set data. A DBCLOB is considered to be a graphic string. It is not supported by the ODBC Driver for DB2.
DECIMAL	SQL_DECIMAL	Yes	A packed decimal number.
DOUBLE	SQL_DOUBLE	Yes	An 8-byte double-precision floating point number.
FLOAT	SQL_FLOAT	Yes	An 8-byte double-precision floating point number. This data type is the same as a DOUBLE.
GRAPHIC (DBCS)	SQL_CHAR	No	A fixed-length graphic string consisting of a sequence of double byte (DBCS only) character string data.
INTEGER	SQL_INTEGER	Yes	A four-byte integer with a precision of 10 digits ranging in value from -2,147,463,648 to +2,147,483,647.
LONG VARCHAR (BIT)	SQL_BINARY	No	A varying-length (double-byte only) character string.
LONG VARCHAR (SBCS)	SQL_CHAR	No	A varying-length SBCS character string.
LONG VARCHAR (MIXED)	SQL_CHAR	No	A varying-length mixed-character (single and double-byte) string.

LONG VARGRAPHIC (DBCS)	SQL_LONGVARCHAR	No	A varying-length graphic string consisting of a sequence of double byte (DBCS only) character string data.
SMALLINT	SQL_SMALLINT	Yes	A SMALLINT (small integer) is a two-byte integer with a precision of 5 digits ranging from -32,768 to +32,767.
REAL	SQL_REAL	Yes	A 4-byte single-precision floating point number.
TIME	SQL_TYPE_TIME	Yes	An 8-byte time string. When using ActiveX Data Objects to return data from a DB2 TIME data type, ADO returns a DATETIME value.
TIMESTAMP	SQL_TYPE_TIMESTAMP	Yes	A 26-byte string representing the date, time, and microseconds.
VARCHAR (BIT)	SQL_BINARY	No	A varying-length (double-byte only) character string.
VARCHAR (SBCS)	SQL_CHAR	Yes	A varying-length character string.
VARCHAR (MIXED)	SQL_CHAR	No	A varying-length mixed (single and double-bye) character string.
VARGRAPHIC (DBCS)	SQL_VARCHAR	No	A varying-length graphic string consisting of a sequence of double byte (DBCS only) character string data.

Not all platforms and versions of DB2 support all of the above-referenced data types. Consult your IBM SQL Reference for the specific target and platform and version of DB2.

The ODBC Driver for DB2 exposes only selected DB2 data types as native types in the ODBC catalog function GetTypeInfo. For example, the driver does not expose LONG CHARACTER or VARYING LONG CHARACTER types. Rather these types are exposed as CHARACTER and VARYING CHARACTER respectively. Also, the driver exposes CHARACTER FOR SBCS DATA, CHARACTER FOR MIXED DATA, and CHARACTER FOR BIT DATA as CHARACTER. The driver exposes VARYING CHARACTER FOR SBCS DATA, VARYING CHARACTER FOR MIXED DATA, and VARYING CHARACTER FOR BIT DATA as VARYING CHARACTER. However, the ODBC Driver for DB2 we will return these LONG and VARYING LONG data types if one reads a table with these data types. For example, when reading a table with a variable character string of length greater than 254 bytes, the ODBC Driver for DB2 will return a LONG VARCHAR.

The ODBC Driver for DB2 can read but not write to columns containing DB2 GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC data types. These types are not exposed as native data types in the ODBC Catalog function GetTypeInfo.

The maximum length of the DB2 character and graphic string data types is dependent on the DB2 platform and version. For example, a CHAR type on DB2 for OS/390 V5R1 has a maximum length of 254 bytes, whereas a CHAR type on DB2/400 V4R4 has a maximum length of 32,766 bytes.

Data conversions from a large numeric type to a small numeric type are supported (from DOUBLE to SINGLE and from INT to SMALLINT, for example), however truncation and conversion errors can occur that will not be reported by the ODBC Driver for DB2.

See the section on Code Page Support Using the ODBC Driver for DB2 for limitations on the support for the DB2 character data types of subtype MIXED using the ODBC Driver for DB2.

Note that the ODBC Driver for DB2 does not support mapping DB2 bit strings and graphic data types to SQL_BINARY. The ODBC Driver for DB2 does not support a binary-to-binary conversion. Consequently, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, and LONGVARCHAR FOR BIT DATA can only be supported when the Process Binary As Character property is set to true when configuring the ODBC data source or passed as part of the connection string. In this way, the ODBC client actually binds the column as an ODBC character type, not as a binary type.

Using the ODBC Driver for DB2, certain conversions of strings from EBCDIC to ASCII and then back to EBCDIC are asymmetric, and can result in strings that are different from the original. The EBCDIC specification contains ordinals for which there is no defined character. The ODBC Driver for DB2 translates all such undefined characters to the question mark character ("?"). So when

ASCII strings containing these characters are converted back to EBCDIC, these undefined characters will be replaced with question marks. To protect EBCDIC strings containing undefined characters, these fields should be tagged as binary strings and mapped by the application.

The ANSI to EBCDIC character conversions affected include the following:

Character Value (Decimal)	Character Value (Hexadecimal)	ANSI Code Page 1252	EBCDIC Character After Conversion to CCSID 37
128	0x80	Not used	?
130	0x82	Single low quote	?
131	0x83	Latin F with hook	?
132	0x84	Double low quote	?
133	0x85	Ellipsis	?
134	0x86	Dagger	?
135	0x87	Double dagger	?
136	0x88	Per mile	?
137	0x89	S with caron	?
138	0x8A	Left angle	?
139	0x8B	Ligature OE	?
140	0x8C	Not used	?
142	0x8E	Not used	?
145-156	0x91-0x9C		?
158-159	0x9E-0x9F		?

Floating Point Considerations Using the ODBC Driver for DB2

When real or double (synonymous with float) data is inserted into a DB2 table as a floating point data type, it is stored in scientific notation. For example, FLOAT(1.1) would be stored as +1.10000E+000.

Care must be taken when executing SQL statements to make sure that the proper data type specified in the SQL statement matches the values stored in DB2. For example, the following select statement would match values in DB2 stored as decimal 1.1

```
SELECT * FROM TEST WHERE C1 = 1.1
```

If the data in DB2 was stored as real numbers, there would not be a match since decimal 1.1 is stored as 1.1, not the representation of +1.10000E+000. When DB2 parses and executes the SQL select statement, it interprets 1.1 as a decimal type. When doing the select query, DB2 does not implicitly do the conversion to floating point. In this case, the SQL statement should explicitly typecast the 1.1 so that DB2 looks for the correct format (the scientific notation format). The select query would look like the following:

```
SELECT * FROM TEST WHERE C1 = REAL(1.1)
```

This will give the results expected. The SQL REAL function will convert the decimal 1.1 to the proper format before DB2 executes the actual select.

Usernames and Passwords Using the ODBC Driver for DB2

When connecting to remote DB2 systems, most users must be authenticated by the remote system by passing a valid user ID and password.

The AS/400 computer is case sensitive with regard to user ID and password; it accepts them only in uppercase. The ODBC Driver for DB2 automatically converts the user ID and password into uppercase when connecting to a DB2 for OS/400 system.

The mainframe is not case sensitive; the user ID and password is acceptable in lowercase or uppercase.

DB2 Universal Database (UDB) for Windows NT is case sensitive; it supports mixed case passwords. Users must enter a password in the correct mixed case. When entering a user ID, use only the Windows NT user name and not the Windows NT domain name.

Errors Returned by the ODBC Driver for DB2

The ODBC Driver for DB2 generates errors in these areas:

- ODBC Driver Manager
- Microsoft ODBC Driver for DB2
- DRDA Application Requester network client

The ODBC Driver Manager is a shared library that establishes connections with ODBC drivers, submits requests to ODBC drivers, and returns results to applications. An ODBC Driver Manager error has the following format:

```
[vendor] [ODBC DLL] message
```

For example:

```
[Microsoft] [ODBC DLL] Driver does not support this function.
```

If you encounter this type of error, check the last ODBC call the application made for possible problems. For further information on ODBC Driver Manager errors, contact your ODBC application vendor, or refer to the ODBC documentation available from Microsoft Press.

An error reported by the Microsoft ODBC Driver for DB2 has the following format:

```
[Microsoft] [ODBC Driver for DB2] message
```

For example:

```
[Microsoft] [ODBC Driver for DB2] Invalid precision specified.
```

If you encounter this type of error is, check the last ODBC call the application made for possible problems. For further information on ODBC Driver errors, contact your ODBC application vendor, or refer to the ODBC documentation available from Microsoft Press.

When using the Microsoft ODBC Driver for DB2, data source refers to the target database. An error that occurs in the data source is returned with the data source name and in the following format:

```
[Microsoft] [ODBC Driver for DB2] [data_source] message
```

For example, an ODBC application may receive the following message from a DB2 data source running on an IBM mainframe:

```
[Microsoft] [ODBC Driver for DB2] [DB2] DB2-0919: specified length too long for CHAR column
```

If you encounter this type of error, the application attempted to perform an operation not supported by the DB2 database system. Check the DB2 database system documentation for more information, or consult your database administrator.

Troubleshooting the ODBC Driver for DB2

The Windows 2000 and Windows NT Event Viewer can be a useful tool for troubleshooting data access in some cases. The ODBC Driver for DB2 does not issue events. However, when SNA (APPC/LU 6.2) is used for the network transport for the ODBC Driver for DB2, the low-level SNA APPC transport issues events on the SNA connection.

The ODBC Driver for DB2 supplied with Host Integration Server 2000 has the ability to trace DRDA data flows when used over TCP/IP. This capability is accessible from the SNADB2 Service tracing inside the Trace utility shipped with Host Integration Server 2000.

This facility shows the same data as an APPC trace but without the control indicators (for example, What_Received). Socket errors are traced and the error codes can be looked up in Winsock2.h supplied with the Win32 SDK.

When the ODBC Driver for DB2 passes an error code, the best source in which to look-up the meaning of the return code is often the SQL Reference or SQL Messages and Codes Reference for the target SQL database. In this case, the target database is one of the DB2 platforms supported by the ODBC Driver for DB2.

The ODBC Driver for DB2 maintains an internal integer variable named SQLCODE and an internal 5-byte character string variable named SQLSTATE used to check the execution of SQL statements on DB2. SQLCODE is set by DB2 after each SQL statement is executed. DB2 returns the following values for SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.
- SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.

SQLSTATE is also set by DB2 after the execution of each SQL statement. Application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE provides application programs with common codes for common error conditions (the values of SQLSTATE are product-specific only if the error or warning is product-specific). Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors.

SQLSTATE values consist of a two-character class code value, followed by a three-character subclass code value. The first character of an SQLSTATE value indicates whether the SQL statement was executed successfully or unsuccessfully (equal to or not equal to zero, respectively). Class code values represent classes of successful and unsuccessful execution conditions. The following SQLSTATE class codes are used by DB2:

Class Code	Description of Error Class
00	Successful completion. Execution of the SQL statement was successful and did not result in any type of warning or exception condition.
01	Warning
02	No data
07	Dynamic SQL error
08	Connection exception
0A	Feature not supported
0F	Invalid token
21	Cardinality violation
22	Data exception
23	Constraint violation
24	Invalid cursor state
25	Invalid Transaction State
26	Invalid SQL statement identifier
2D	Invalid transaction termination
34	Invalid cursor name
39	External function call exception
40	Transaction rollback
42	Syntax error or access rule violation
44	WITH CHECK OPTION violation
51	Invalid application state

53	Invalid operand or inconsistent specification
54	SQL or product limit exceeded
55	Object not in prerequisite state
56	Miscellaneous SQL or product error
57	Resource not available or operator intervention
58	System error

The SQLSTATE value of HY000 is defined as a driver-specific error. An SQLSTATE of 08S01 (connection exception with a subclass code of S01) also indicates a driver-specific error. This means the SQLCODE should be looked up in the driver-specific documentation included with the ODBC Driver for DB2.

If the SQLSTATE does not indicate a driver-specific error when the ODBC Driver for DB2 passes back an SQLSTATE of 08S01, it indicates a network error. For example, an SQLCODE of -603 is a driver-specific error that is mapped to DB2OLEDB_COMM_HOST_CONNECT_FAILED in the db2oledb.h include file supplied with the ODBC Driver for DB2. Errors with an SQLSTATE of 08S01 are documented in the db2oledb.h include file (the SQLCODE value) which is located on the Host Integration Server 2000 CD-ROM in the SDK\Include subdirectory.

The following steps are useful in researching an error. Start by reading the provided error text returned by the ODBC Driver for DB2. In some cases, the error text may provide limited information. For example, error text from an SQLCODE of -603 reads:

```
Test connection failed because of an error in initializing driver.
Could not connect to specified host.
```

The next step is to lookup the SQLSTATE to determine the source of the error. Is the error a DB2 error, a network client error, or an ODBC Driver error? An SQLSTATE of 08S01 is defined as follows:

```
Communication link failure.
```

This definition is intended to inform the user, administrator, or developer that the error is one related to the ODBC driver's underlying network client.

Unfortunately, many of the SQLSTATE codes returned by the ODBC Driver for DB2 are DB2 errors and are not documented in the ODBC Driver for DB2 online Help.

The SQLSTATE of HY000 is defined as a driver-specific error. An SQLSTATE of 08S01 also indicates a driver-specific error. In this case, you should look up the SQLCODE in the driver-specific documentation included with the ODBC Driver for DB2.

If the SQLSTATE does not indicate a driver-specific error, you should look up the SQLCODE in the appropriate DB2 manual for the target platform. For example, an SQLCODE of -603 is documented in Appendix B, "SQLCODEs and SQLSTATEs," in the *AS/400 Advanced Series DB2 for AS/400 SQL Programming, Version 4*, document number SC41-5611-00 published by IBM. An SQLCODE of -603 corresponds to SQLSTATE 23515 in the DB2 for OS/400 error code list. For example, the explanation for this SQLCODE is:

```
Unique index cannot be created because of duplicate keys.
```

When the SQLSTATE and the SQLCODE definitions documented in these appendixes create a mismatch with the actual errors returned, it usually indicates a driver-specific error condition.

A final step in understanding an error is to check the db2oledb.h file. This file is not installed by Setup for the Host Integration Client 2000, but is located on the CD-ROM for in the SDK\Include folder. An SQLCODE (for example, -603) can be found by searching the rightmost column of the db2oledb.h file for a value near to 603. For instance, locate the comment "/* -600 */" and then count down three additional lines to line number 603. The internal error code -603 is defined as follows:

```
DB2OLEDB_COMM_HOST_CONNECT_FAILED.
```

Unfortunately, this error text is not further defined anywhere in the software or documentation provided to the customer. This particular error usually indicates a problem with the configuration parameters or the connection string passed.

ActiveX Controls

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about how to integrate your applications using the Data Queue and Host File Transfer ActiveX® controls.

This section contains:

Using the Data Queue ActiveX Control

A data queue is an AS/400 system object that is used for inter-process communications between multiple programs or jobs. Data queues allow multiple programs to send and receive shared messages via a central repository without first writing the message data to a physical database file. Typically, when a data record is read from the queue, the record is erased from the queue. The advantage of using data queues to share data in comparison with using database files is that data queues require much less file I/O and therefore improve overall system performance.

The Microsoft® Data Queue ActiveX® Control provides the ability to access AS/400 data queues. Host Integration Server 2000 provides this service via a single ActiveX Control that depends on other core Host Integration Server DLLs. Developers can move part or all of their AS/400 applications from an AS/400 computer to a PC platform, while retaining access in the program running on the PC to a remote data queue on the AS/400.

The Microsoft Data Queue ActiveX Control is implemented as a Distributed Data Management (DDM) Application Requester. The Data Queue ActiveX Control uses the Data Queue interfaces in the DDM Level 4 architecture, which are extensions to the record-level input/output (RLIO) protocol of IBM's Distributed Data Management architecture.

DDM is a set of rules for distributing or extending data management from one computer to another, such as from a mainframe to an AS/400 computer, or from one of these host computers to a server computer. By combining the Microsoft Data Queue ActiveX Control and DDM architectures, Microsoft enables organizations to preserve their investments in existing data management infrastructure, while extending universal file and data transfer to all enterprise-wide data sources.

The information in this section is required to develop applications with Host Integration Server 2000 that use ActiveX or COM objects to transfer data from local machines to AS/400 Data Queues in a Systems Network Architecture (SNA) environment or over TCP/IP using RLIO and DDM.

This section contains:

Advantages of Data Queues

Data queues provide a fast means of inter-process communication, requiring low system overhead and minimal setup. AS/400 Data Queues are designed to provide a flexible, highly efficient, yet temporary means of inter-process communication. Data queues are familiar to most AS/400 programmers as a simply method of passing information to another program.

Data queues provide considerable flexibility to the application programmer. The data queues interfaces require no communications programming and can be used either for connected or disconnected communication. AS/400 and PC applications can be developed using any supported language, yet still communicate with each other. PC programs can communicate with AS/400 programs via a common AS/400 data queue. The use of data queues requires little knowledge of communication and no knowledge of APPC if the programmer utilizes the Microsoft® Data Queue ActiveX® control. The data queue messages are merely described at the record-level, allowing the application programmer to define the field-level structure as required.

By default, when one program reads an entry in the queue, the entry is then deleted. Pointers to the queue entries are then updated to reflect the change in the record stack. A data queue can exist with no entries, a single entry, or multiple entries. Multiple concurrent jobs and programs can access data queues.

When receiving data, the requesting application can set a timeout value to wait for data to arrive in the queue. Waits can be applied based on entry of the data record or for a time period (zero seconds to many days in length). A program that reads from a queue need not be running when the queue is created or when records are inserted. A single data queue can support many separate interactive jobs. At regular intervals or at the end of the day, records in the data queue can be persisted to a file by a single automated batch process.

Platforms Supported by the Data Queue ActiveX Control

AS/400 Data Queue support in Host Integration Server 2000 is implemented by extending the features of the existing Distributed Data Management (DDM) Application Requester and by the creation of an ActiveX control. To support data queues via DDM, a large number of DDM commands are implemented. DDM Data Queue support requires a DDM Architecture Level 4 implementation for both source and target.

The Data Queue ActiveX® Control requires the following system software on the AS/400:

- OS/400 V3R7 or higher.
- OS/400 V3R2 or higher.
- OS/400 V3R0M5 with the following program technical fixes applied: SF21521, SF21498, SF21500, and SF21254.
- OS/400 V3R1M0 with the following program technical fixes applied: SF21555, SF21499, SF21501, and SF21266.
- OS/400 V2R3 with the following program technical fixes applied: SF21522, SF19749, SF19748, and SF19122.

The Data Queue ActiveX Control supplied with Host Integration Server 2000 supports the following operating systems:

- Microsoft® Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Data Center
- Microsoft Windows 2000 Professional
- Microsoft Windows NT® Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 6a or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later
- Microsoft Windows 98, Second Edition

The Data Queue ActiveX Control supplied with Host Integration Server 2000 Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

The Data Queue ActiveX Control requires the following computer-to-host connectivity software:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server 2000 Administrator Client
- Microsoft Host Integration Server 2000 End-User Client

Registry Settings Used by Data Queues

The Microsoft® Data Queue ActiveX® Control uses a number of registry settings for configuration and proper operation. The configuration registry settings are located under the **HKEY_LOCAL_MACHINE\Software\Microsoft\SNA Server\CurrentVersion\Setup** key. These registry settings include the following subkeys:

Subkey	Comment
RootDir	Stores the path to root directory where the Host Integration Server was installed. The system directory below this root directory is the location where the Data Queue ActiveX Control DLL and other support DLLs are installed.

Object Support Using Data Queues

The Microsoft® Data Queue ActiveX® Control supports a number of standard COM interfaces as well some custom objects and interfaces.

COM Interface Support Using Data Queues

The Microsoft Data Queue ActiveX Control supports a number of standard COM interfaces as well as several custom interface, **IEIGDataQueueCtl**, **IEIGDataQueue**, and **IEIGDataQueueItem**. The ActiveX Control object has the ability to register and de-register itself via standard control mechanisms. Support for a number of standard COM interfaces makes it easy to develop applications using the Data Queue ActiveX Control with Microsoft Visual Basic® and Microsoft Visual C++® as well as from Microsoft Internet Explorer and Microsoft Access. Supporting a variety of standard COM interfaces also provides different ways for a client to save information.

The following table summarizes the standard COM interfaces supported by the Data Queue ActiveX Control.

COM Interface	Comments
ICategorizeProperties	This interface divides up the properties into an intelligent presentation to the client.
IConnectionPointContainer	
IDispatch	A dual interface deriving from **IDispatch** is exposed to provide support and flexibility to clients. Clients that provide support for automation interfaces will utilize the **IDispatch** interface, while more robust clients may use the custom interface. Using the custom interface provides for the greatest execution speed.
IPerPropertyBrowsing	This interface provides support for client browsing of properties in an intelligent manner. This interface exposes to the client property lists used in the population of a dropdown list. This interface is required for the control to be hosted by Microsoft Access.
IPropertyNotifySink	The interface is implemented by a sink object to receive notifications about property changes from an object that supports **IPropertyNotifySink** as an outgoing interface. The client that needs to receive the notifications in this interface (from a supporting connectable object) creates a sink with this interface and connects it to the connectable object through the connection point mechanism.
IPersist	
IPersistPropertyBag	This interface is the preferred method of property persisting for Internet Explorer and Visual Basic. Using this interface persisted properties are stored as a set of name/VARIANT value pairs.
IPersistStorage	This interface stores persistent properties into a structured storage object.
IPersistStreamInit	This interface is responsible for saving the persisted properties in binary form using a stream interface. This is the method used by the Microsoft Visual C/C++ compiler to persist properties.
ISupportErrorInfo	This interface is the preferred method to return error indications to scripting clients. Using this interface, error codes and explanation text are returned to the client. This information may be used in order to provide diagnostic information to the user and in cases of failure.

The IEIGDataQueueCtl Object

The **IEIGDataQueueCtl** object supports a number of properties and methods that provide the ability to connect with a host and communicate with OS/400 Data Queues. The **IEIGDataQueueCtl** also supports a set of events notifying a client application of connection status and error reporting. These events are handled by the client supporting several callback functions and setting these callbacks using the **IConnectionPointContainer**.

The following **IEIGDataQueueCtl** object methods are supported by the Microsoft Data Queue ActiveX Control.

Method Name	Comment
Connect method	Establishes a connection to the configured host and reports to the user an indication of the success or failure of the action.
CreateQueueContainer method	Create an instance of a IEIGDataQueue container object and optionally initialize the QueueName property. The created queue object is assumed to be associated with the connection object that created it for the life of the connection or the life of the queue object.
Disconnect method	Terminates an existing connection to a host machine.

The following **IEIGDataQueueCtl** object properties are supported by the Microsoft Data Queue ActiveX Control.

Property Name	Comment
CCSID property	Sets or returns the character code set identifier (CCSID) that must match the data in the AS/400 data queue as represented on the remote host computer. This property defaults to U.S./Canada (37).
ConnectionState property	Returns the current state of the connection. The state of a connection can be unspecified, idle, connecting, connected, or disconnecting.
ConnectionType property	Sets or returns the network transport used for this connection. The **ConnectionType** property designates whether the Data Queue ActiveX Control connects via APPC (SNA LU6.2) or TCP/IP. The possible values for this parameter are TCPIP or APPC. The default value for this parameter is SNA. If APPC is selected, then values for the **LocalLU**, **ModeName**, and **RemoteLU** properties are required. If TCPIP is selected, then values for **NetAddr** and **NetPort** properties are required. Note that a TCP/IP connection is not currently supported when connecting to an AS/400 using the Data Queue ActiveX Control, since TCP/IP is not supported by the AS/400 DDM implementation.
LocalLU property	Sets or returns the Local LU Alias. When APPC (SNA LU 6.2) is selected for the ConnectionType property, this property must match the name of the local LU alias configured using SNA Manager. This property defaults to the string value of "LOCAL" represented as a BSTR.
ModeName property	Sets or returns the APPC mode. When APPC (LU 6.2 SNA) is selected for the **ConnectionType** property, this field must be set to the APPC mode that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This property defaults to the string value of "QPCSUPP" represented as a BSTR.
NetAddr property	Sets or returns the IP address of the host computer. When TCPIP (a TCP/IP connection) has been selected for the **ConnectionType** property, this property indicates the IP address of the host. This property can be an IP address or the name representing the host IP address using the Domain Name System (sna.microsoft.com, for example). This property is a string (BSTR) and has no default value. Note that a TCP/IP connection is not currently supported when connecting to an AS/400 using the Data Queue ActiveX Control, since TCP/IP is not supported by the AS/400 DDM implementation.

NetPort property	Sets or returns the TCP/IP port used for communication with the host. When TCPIP (a TCP/IP connection) has been selected for the **ConnectionType** property, this parameter is the TCP/IP port used for communication with the host. The default value for this property is the string (BSTR) "446" representing TCP/IP port 446. Note that a TCP/IP connection is not currently supported when connecting to an AS/400 using the Data Queue ActiveX Control, since TCP/IP is not supported by the AS/400 DDM implementation.
Password property	Sets or returns the password used for authentication. A valid user name and password are normally required to access data on a host computer. The password is case sensitive and is normally displayed as asterisks in a dialog box for security purposes. This property is a string (BSTR) and has no default value.
PCCodePage property	Sets or returns the PC code page The PC Code Page property indicates the code page to be used on the PC for character code conversion. This property defaults to Latin 1 (1252).
RemoteLU property	Sets or returns the Remote LU Alias. When APPC (LU 6.2 SNA) is selected for the **ConnectionType** property, this property is the name of the remote LU alias configured using SNA Manager. This property is a string (BSTR) and has no default value.
UserID property	Sets or returns the username used for authentication. A valid user name and password are normally required to access data on a host computer. This value is case sensitive. This property is a string (BSTR) and has no default value.

The IEIGDataQueue Object

The **IEIGDataQueue** object represents a logical queue and supports a number of properties and methods that provide the ability to communicate with a specific data queue. The QueueName property is the name of the physical queue. All methods and events are related to the queue that is represented by the individual instance of the object. The Data Queue ActiveX Control also supports a set of events notifying a client application of connection status, data transfer status, and error reporting. These events are handled by the client supporting several callback functions and setting these callbacks using the **IConnectionPointContainer**.

The following **IEIGDataQueue** object methods are supported by the Microsoft Data Queue ActiveX Control.

Method Name	**Comment**
AddQueueItem method	Adds a record to the current queue.
Cancel method	Terminates a request to receive information from the queue that is already in progress.
CancelQueue method	Indicates that an application no longer wants to be notified of an incoming queue data item. This can be used to stop pending notifications that were queued as a result of calling **GetQueueItem**.
ClearAll method	Removes all items from the queue.
CreateQueue method	Creates a data queue.
DeleteQueue method	Clears all messages from the queue and then deletes the queue.
Get_QueueName method	Retrieves the queue name from a data queue.
GetQueueItem method	Retrieves an item from the queue.
QueryAttribute method	Requests information on one of the queue's attributes
SetAttribute method	Changes the attributes associated with a data queue.
StopQueue method	Stops the queue from responding to client requests.

The following **IEIGDataQueue** object property is supported by the Microsoft Data Queue ActiveX Control.

Property Name	**Comment**
QueueName property	This is the name of the data queue this object is associated with.

The IEIGDataQueueItem Object

The **IEIGDataQueueItem** object represents a specific queue item and supports a number of properties and methods.

The following **IEIGDataQueueItem** object method is supported by the Microsoft Data Queue ActiveX Control.

Method Name	Comment
Reset method	Resets the queue item properties to default values.

The following **IEIGDataQueueItem** object properties are supported by the Microsoft Data Queue ActiveX Control.

Property Name	Comment
ExtUser property	The external job user name.
ExtJobName property	The external job name.
ExtJobNumber property	The external job number.
InactiveRec property	Indicates an Inactive record.
Keyval property	The key value.
Message property	The queue message.
QItemType property	The type of queue item this represents.
Record property	The entire queue data.
RecordAttribute property	The list of record attributes.
RecCount property	The record count.
RecNumber property	The record number.
ReplyRequest property	Indicates if the reply message should be returned.
UsrProf property	The user profile.

IEIGDataQueueCtlEvents Notifications

The **IEIGDataQueueCtl** object of the Microsoft Data Queue ActiveX Control also supports a set of events notifying a client application of connection status and error reporting. These events are handled by the client supporting several callback interfaces and setting these callbacks derived from the standard **IConnectionPointContainer** COM object.

The following **IEIGDataQueueCtlEvents** notification interface methods are supported by the Data Queue ActiveX Control:

Event Notifications	Comment
ConnectionStateChange	This event is fired when the state of a connection has changed. A ConnectionState parameter is passed to the client callback function that receives this event method call. This parameter is an eigConnectionStateEnum value representing the new state of the ConnectionState property.
ReportError	This event is fired when an error condition needs to be reported during non-blocking (asynchronous) functions. This event passes two parameters to the client callback function that receives this event method call. The first parameter is a long value representing an error code. The second parameter is a BSTR string containing a brief text description of the error.

IEIGDataQueueEvents Notifications

The **IEIGDataQueue** object of the Microsoft Data Queue ActiveX Control also supports a set of events notifying a client application of when transfers are completed, requests are received, and error reporting. These events are handled by the client supporting several callback interfaces and setting these callbacks derived from the standard **IConnectionPointContainer** COM object.

The following **IEIGDataQueueEvents** notification interface methods are supported by the Data Queue ActiveX Control:

Event Notifications	Comment
ReportError2	This event is fired when an error condition needs to be reported during non-blocking (asynchronous) functions. This event passes two parameters to the client callback function that receives this event method call. The first parameter is a long value representing an error code. The second parameter is a BSTR string containing a brief text description of the error.
RequestReceived	This event is fired when a request is received.
SendComplete	This event is fired as an indication to the client that the requested transfer operation has completed.

Programming Considerations Using the Data Queue ActiveX Control

The Microsoft® Data Queue ActiveX® Control exposes a dual interface deriving from **IDispatch**. This provides support and flexibility to clients wishing to use the object. Clients that provide support for automation interfaces can use the **IDispatch** interface while more robust clients may use the custom interface. Using the custom interface offers the greatest execution speed.

The single-threading model is supported, allowing only single threads to access the objects safely.

Asynchronous read operations are not currently supported. The BlockComplete parameter of the **GetQueueItem** method must be set to a value of 0 (eigAnswerYes), indicating that the **GetQueueItem** operation should block until the completion status is known.

Code Page Support Using Data Queues

When using the Data Queue ActiveX Control, the Host CCSID (character code set identifier) property should be configured to match the data as represented on the remote host computer. The Host CCSID parameter defaults to EBCDIC U.S./Canada (37) when using the Data Queue ActiveX Control.

DBCS Code Page Support Using Data Queues

Support for Double-Byte Character String (DBCS) data is limited using the Data Queue ActiveX Control. Conversions between DBCS and ANSI code pages are not supported. Conversions between DBCS and ISO code pages are not supported.

The DB2 GRAPHIC data types (GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC) are not supported. These DB2 data types support DBCS (not mixed) data. Mixed data types are supported using CHAR FOR MIXED DATA, VARCHAR FOR MIXED DATA, and LONGVARCHAR FOR MIXED DATA.

Usernames and Passwords Using Data Queues

When connecting to host systems, most users must be authenticated by the remote system by passing a valid User ID and Password.

The AS/400 computer is case sensitive with regard to User ID and Password. The AS/400 only accepts a User ID and Password in uppercase. The Microsoft Data Queue ActiveX Control will force the User ID and Password into uppercase when it knows that it is connecting to an AS/400 system.

Troubleshooting the Data Queue ActiveX Control

The Microsoft Data Queue ActiveX Control supplied with Host Integration Server 2000 has the ability to trace DRDA data flows when used over TCP/IP.

This tracing capability is accessible from the SNADB2 Service tracing inside the Trace tool. This facility will show the same data as an APPC trace but without the control indicators (For example, What_Received). Socket errors are traced and the error codes can be looked up in Winsock2.h supplied with the Platform SDK.

The Data Queue ActiveX Control can return the following types of errors:

- Errors from the remote hosts
- Microsoft Data Queue-specific errors
- Errors from the underlying DDM Application Requester network client

Using the Host File Transfer ActiveX Control

The Microsoft® Host File Transfer ActiveX Control provides the ability to transfer files between a local machine and an MVS, OS/390, AS/400, or AS/36 host system. Host Integration Server 2000 provides this service via a single ActiveX Control that depends on other core Host Integration Server DLLs. This extends the ability for a client application to perform file transfer operations from a large number of client development environments.

The Microsoft Host File Transfer ActiveX Control uses the record-level input/output (RLIO) protocol of IBM's Distributed Data Management (DDM) architecture to transfer files. The Host File Transfer ActiveX Control is implemented as a Distributed Data Management (DDM) source requester, which communicates via APPC LU6.2 or TCP/IP to a DDM target server.

DDM is a set of rules for distributing or extending data management from one computer to another, such as from a mainframe to an AS/400 computer, or from one of these host computers to a server computer. By combining the Microsoft File Transfer ActiveX Control and DDM architectures, Microsoft enables organizations to preserve their investments in existing data management infrastructure, while extending universal file transfer to all enterprise-wide data sources.

The information in this section is required to develop applications with Host Integration Server that use ActiveX or COM objects to transfer files from local machines to hosts in a Systems Network Architecture (SNA) environment or over TCP/IP using RLIO and DDM.

This section covers the following topics about the Microsoft Host File Transfer ActiveX Control:

This section contains:

Platforms Supported by the Host File Transfer ActiveX Control

On the mainframe platform, IBM offers a target DDM server implementation in IBM Distributed File Manager (DFM), a component of IBM Data Facility Storage Management Subsystem (DFSMS). The Microsoft® Host File Transfer ActiveX Control requires DFSMS version 1 release 2 or later for MVS/ESA and OS/390 to support an SNA LU6.2 connection.

On midrange AS/400 computers, IBM has implemented target DDM servers directly in OS/400. The Microsoft Host File Transfer ActiveX Control requires OS/400 Version 3 Release 2 or later to support an SNA LU6.2 connection. The Microsoft Host File Transfer ActiveX Control requires OS/400 Version 4 Release 2 or later to support a TCP/IP connection.

On the AS/400 platform, the Host File Transfer ActiveX Control supports physical and logical files with an associated external record description file. For specific limitations, please see the *AS/400 DDM User's Guide*.

On the mainframe platform, the Host File Transfer ActiveX Control supports the following data set types:

Sequential Access Method (SAM) data sets

- Basic Sequential Access Method data sets (BSAM)
- Queued Sequential Access Method data sets (QSAM)

Basic Partitioned Access Method (PDS) data sets

- Partitioned Data Set Extended members (PDSE)
- Partitioned Data Set members (PDS)

Virtual Storage Access Method (VSAM) data sets

- Entry-Sequenced Data Sets (ESDS)
- Key-Sequenced Data Sets (KSDS)
- Fixed-Length Relative Record Data Sets (RRDS)
- Variable-Length Relative Record Data Sets (VRRDS)
- Relative Record Data Set (RRDS)
- VSAM Alternate Indexes for ESDS and KSDS data sets

The preceding data set types are supported by IBM DFM/MVS. The following data set types are not supported by DFM/MVS and cannot be accessed using the Host File Transfer ActiveX Control.

- VSAM Linear Data Sets (LDS)
- Generation Data Groups (GDG)
- Generation Data Sets (GDS)
- Basic Direct Access Method data sets (BDAM)
- Indexed Sequential Access Method data sets (ISAM)
- Sequential Data Striping data sets
- OpenEdition MVS Hierarchical File System (HFS) files
- Tape Media

All mainframe data sets accessible through IBM Distributed File Manager must be cataloged in an Intersystem communications function (ICF) catalog and reside on direct access storage devices (DASD).

The Host File Transfer ActiveX Control supplied with Host Integration Server 2000 supports the following operating systems:

- Microsoft® Windows® 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Professional
- Microsoft Windows NT® Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 6a or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later
- Microsoft Windows 98, Second Edition

The Host File Transfer ActiveX Control supplied with Host Integration Server 2000 Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft® Windows® Millennium Edition

The Host File Transfer ActiveX Control requires the following computer-to-host connectivity software:

- Microsoft Host Integration Server 2000
- Microsoft Host Integration Server 2000 Administrator Client
- Microsoft Host Integration Server 2000 End-User Client

Configuring Data Descriptions for Host File Transfer

In order to use the Microsoft Host File Transfer ActiveX Control to transfer files, a user or client application must describe the data format of the host file to transfer. A host data description is normally configured using the Data Descriptions tool.

Microsoft® Management Console (MMC) and MMC snap-ins are the current method of exposing administrative tasks and options in server-based Microsoft products. An MMC snap-in for Data Integration is installed with the Host Integration Client 2000, which enables you to configure data descriptions for transferring files on OS/390, OS/400, MVS/ESA, and AS/36systems. The Data Integration Management Console snap-in enables you to configure data descriptions used by the Host File Transfer ActiveX Control.

The **Data Integration** console contains one high-level object:

- **Data Descriptions**—Stored in Host Column Description (HCD) files that contain the information required to convert host data types to PC computer data types.

When creating a Data Description for use with the Host File Transfer ActiveX Control, the **Use Table for File Transfer** checkbox must be checked and values must be entered for the following additional parameters:

Parameter	Comment
FieldDelimiter	This value represents the delimiter used to mark the end of one field and the beginning of another within a single record. The position of the field elements within a record is assumed to be absolute as configured and variable length fields are not supported. This element is used in order to inform the conversion routine to remove this character if it is found at the position within the record as indicated by the Data Description. This parameter is required and has no default value. The comma character "," or tab character "\t" is commonly used with desktop applications as a field separator.
RecordDelimiter	This value represents the character or characters that appear at the ending of a record. This element is used in order to inform the conversion routine to remove this character if it is found in the last position of the record as indicated by the Data Description. This parameter is required and has no default value. The end-of-line character sequence is commonly used as a record delimiter. The carriage return and linefeed character sequence "\n\r" is commonly used with desktop applications. The newline character "\n" is standard for use on UNIX systems and with some desktop applications.
TextQualifier	This value is used in order to allow elements that may contain the **FieldDelimiter** character to be properly parsed by the parsing engine. If this element is enabled on a particular field, then the inclusion of this character at the beginning of the field is an indication to the parsing engine that all characters up to the next instance of the **TextQualifier** should be treated as part of the current field being processed. This parameter has no default value. The single quote or double quote character is sometimes used as a text qualifier to protect a comma character included in a field from being misinterpreted as a field delimiter.

The **Data Integration** console and the **Data Descriptions** tool is designed to run on Microsoft® Windows 2000, Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95. On Windows 2000 and Windows NT, the console respects the Windows 2000 and Windows NT security hierarchy, where only privileged users can read and write to some areas of the system registry and the file system. To prevent general users from modifying the HCD files on Windows 2000 and Windows NT, the Data Descriptions tool can only be run by users that have administrative privileges on the local computer.

Registry Settings Used By Host File Transfer

The Microsoft® Host File Transfer ActiveX Control uses a number of registry settings for configuration and proper operation. The configuration registry settings are located under the **HKEY_LOCAL_MACHINE\Software\Microsoft\SNA Server\CurrentVersion\Setup** key. These registry settings include the following subkey:

Subkey	Comment
RootDir	Stores the path to root directory where the Host Integration Server was installed. The system directory below this root directory is the location where the Host File Transfer ActiveX Control DLL and other support DLLs are installed.

Object Support Using Host File Transfer

The Microsoft® Host File Transfer ActiveX Control supports a number of standard COM interfaces as well some custom objects and interfaces.

COM Interface Support Using Host File Transfer

The Microsoft® Host File Transfer ActiveX Control supports a number of standard COM interfaces as well as a single custom interface, **IEIGFileTransferCtl**. The ActiveX Control object has the ability to register and de-register itself via standard control mechanisms. Support for a number of standard COM interfaces makes it easy to develop applications using the Host File Transfer ActiveX Control with Visual Basic and Visual C++ as well as from Microsoft Internet Explorer and Microsoft Access. Supporting a variety of standard COM interfaces also provides different ways for a client to save information.

The following table summarizes the standard COM interfaces supported by the Host File Transfer ActiveX Control.

COM Interface	Comments
ICategorizeProperties	This interface divides up the properties into an intelligent presentation to the client.
IConnectionPointContainer	
IDispatch	A dual interface deriving from **IDispatch** is exposed to provide support and flexibility to clients. Clients that provide support for automation interfaces will utilize the **IDispatch** interface, while more robust clients may use the custom interface. Using the custom interface provides for the greatest execution speed.
IOleControl	
IOleInPlaceActiveObject	
IOleInPlaceObject	
IOleInPlaceObjectWindowless	
IOleObject	
IOleWindow	
IPerPropertyBrowsing	This interface provides support for client browsing of properties in an intelligent manner. This interface exposes to the client property lists used in the population of a dropdown list. This interface is required for the control to be hosted by Microsoft Access.
IPropertyNotifySink	The interface is implemented by a sink object to receive notifications about property changes from an object that supports **IPropertyNotifySink** as an outgoing interface. The client that needs to receive the notifications in this interface (from a supporting connectable object) creates a sink with this interface and connects it to the connectable object through the connection point mechanism.
IPersist	
IPersistPropertyBag	This interface is the preferred method of property persisting for Internet Explorer and Visual Basic. Using this interface, persisted properties are stored as a set of name/VARIANT value pairs.
IPersistStorage	This interface stores persistent properties into a structured storage object.
IPersistStreamInit	This interface is responsible for saving the persisted properties in binary form using a stream interface. This is the method used by the Microsoft Visual C/C++ compiler to persist properties.
IProvideClassInfo	

IProvideClassInfo2	
IQuickActivate	
ISupportErrorInfo	This interface is the preferred method to return error indications to scripting clients. Using this interface, error codes and explanation text are returned to the client. This information may be used in order to provide diagnostic information to the user and in cases of failure.
IViewObject	
IViewObject2	
IViewObjectEx	

The IEIGFileTransferCtl Object

The Microsoft® Host File Transfer ActiveX Control supports a number of standard COM interfaces as well as a single custom interface. The **IEIGFileTransferCtl** object supports a number or properties and methods that provide the ability to transfer files to and from MVS, OS/390, AS/400, or AS/36 hosts. The Host File Transfer ActiveX Control also supports a set of events notifying a client application of connection status, file transfer status, and error reporting. These events are handled by the client supporting several callback functions and setting these callbacks using the IConnectionPointContainer.

The following **IEIGFileTransferCtl** object methods are supported by the Microsoft Host File Transfer ActiveX Control:

Method Name	Comment
Cancel method	Terminate a file transfer operation that is already in progress.
Connect method	Establishes a connection to the configured host and reports to the user an indication of the success or failure of the action.
Disconnect method	Terminates an existing connection to a host machine.
GetFile method	Copy a file from host storage to local storage. This method requires the two file names as parameters.
PutFile method	Copy a file from local storage to host storage. This method requires the two file names as parameters.

The following **IEIGFileTransferCtl** object properties are supported by the Microsoft Host File Transfer ActiveX Control:

Property Name	Comment
AppendToEnd property	Sets or returns whether a file transfer should append to the end of a file (eigAnswerYes) if the file exists, or should it overwrite the existing contents replacing the data with the new information (eigAnswerNo). This property defaults to eigAnswerYes (0).
CCSID property	Sets or returns the character code set identifier (CCSID) that must match the data in the file as represented on the remote host computer. This property defaults to U.S./Canada (37).
ConnectionState property	Returns the current state of the connection. The state of a connection can be unspecified, idle, connecting, connected, or disconnecting.
ConnectionType property	Sets or returns the network transport used for this connection. The **ConnectionType** property designates whether the Host File Transfer ActiveX Control connects via APPC (SNA LU6.2) or TCP/IP. The possible values for this parameter are a TCP/IP or an APPC connection using an enumerated value. The default value for this parameter is an APPC (SNA) connection type. If APPC is selected, then values for the **LocalLU**, **ModeName**, and **RemoteLU** properties are required. If TCP/IP is selected, then values for **NetAddr** and **NetPort** properties are required.
CreateIfNonExisting property	Sets or returns whether a file operation should create a new destination file if one does not already exist (eigAnswerYes). This property defaults to eigAnswerNo (1)
LocalLU property	Sets or returns the Local LU Alias. When LU 6.2 (SNA) is selected for the ConnectionType property, this property must match the name of the local LU alias configured using SNA Manager. This property defaults to the string value of "LOCAL" represented as a BSTR.
ModeName property	Sets or returns the APPC mode. When APPC (LU 6.2 SNA) is selected for the ConnectionType property, this field must bet set to the APPC mode that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This property defaults to the string value of "QPCSUPP" represented as a BSTR.

NetAddr property	Sets or returns the IP address of the host computer. When TCP/IP has been selected for the **ConnectionType** property, this property indicates the IP address of the host. This property can be an IP address or the name representing the host IP address using the Domain Name System (sna.microsoft.com, for example). This property is a string (BSTR) and has no default value.
NetPort property	Sets or returns the TCP/IP port used for communication with the host. When TCP/IP has been selected for the ConnectionType property, this parameter is the TCP/IP port used for communication with the host. The default value for this property is the string (BSTR) "446" representing TCP/IP port 446.
OverwriteHostFile property	Sets or returns whether a file operation request to copy a file that will write over an existing file will fail. When this property is set to eigAnswerNo, a request to write a file over an existing file will fail. This property defaults to eigAnswerNo (1)
Password property	Sets or returns the password used for authentication. A valid user name and password are normally required to access files on a host computer. The password is case sensitive and is normally displayed as asterisks in a dialog box for security purposes. This property is a string (BSTR) and has no default value.
PCCodePage property	Sets or returns the PC codepage The PC Code Page property indicates the code page to be used on the PC for character code conversion. This property defaults to Latin 1 (1252).
RDBName property	Sets or returns the name of the remote database name and the Host Column Description (HCD) file that describes the data types and data conversions used to transfer this file. The HCD file describing the data should be located in the system subdirectory below the root directory where Host Integration Server was installed. Setup defaults to the following location: C:\Program Files\Host Integration Server When TCP/IP is selected for the **ConnectionType** property, the **RDBName** must also match the name of the remote database system.
RemoteLU property	Sets or returns the Remote LU Alias. When APPC (LU 6.2 SNA) is selected for the **ConnectionType** property, this property is the name of the local LU alias configured using SNA Manager. This property is a string (BSTR) has no default value.
UserID property	Sets or returns the username used for authentication. A valid user name and password are normally required to access files on a host computer. This value is case sensitive. This property is a string (BSTR) and has no default value.

IEIGFileTransferCtlEvents Notification

The Microsoft® Host File Transfer ActiveX Control also supports a set of events notifying a client application of connection status, file transfer status, and error reporting. These events are handled by the client supporting several callback interfaces and setting these callbacks derived from the standard **IConnectionPointContainer** COM object.

The following **IEIGFileTransferCtlEvents** notification interface methods are supported by the Microsoft Host File Transfer ActiveX Control:

Event Notifications	Comment
ConnectionStateChange	This event is fired when the state of a connection has changed. A ConnectionState parameter is passed to the client callback function that receives this event method call. This parameter is an eigConnectionStateEnum value representing the new state of the ConnectionState property.
ReportError	This event is used in order to return error conditions that occur during the synchronous processing of methods. This event passes two parameters to the client callback function that receives this event method call. The first parameter is a long value representing an error code. The second parameter is a BSTR string containing a brief text description of the error.
TransferComplete	This event is an indication to the client that the requested transfer operation has completed.
TransferProgress	This event will be fired periodically in order to inform the client of the progress of an unattended file transfer. A PercentageDone parameter is passed to the client callback function that receives this event method call. This parameter is a short value representing the percentage complete of the requested operation ranging from 0 to 100. A client application has the option to terminate a file transfer when this event is fired.

Programming Considerations Using Host File Transfer

The Host File Transfer ActiveX Control exposes a dual interface deriving from **IDispatch**. This provides support and flexibility to clients wishing to use the object. Clients that provide support automation interfaces can use the **IDispatch** interface while more robust clients may use the custom interface. Using the custom interface offers the greatest execution speed.

The single-threading model is supported, allowing only single threads to access the objects safely.

The Host File Transfer ActiveX Control does not support uploading Direct Relative Record Data Set (RRDS) files on System/36. An error (381) will occur and the following error message will be received.

```
"Target Not Supported"
```

The Host File Transfer ActiveX Control also does not support uploading RRDS files with the CreateIfNonExisting property option set to yes on System/36. An error will occur (381) and the following message will be received.

```
"Target Not Supported"
```

If there is existing data in a file on OS/390, setting the OverwriteHostFile and AppendToEnd properties to "no" should cause an error (58) to occur and the following error message should be received.

```
"File contains existing data. Upload not configured to append data - upload aborted"
```

On OS/390, this error is not triggered for sequential or KSDS files. Instead the data is appended to the existing data in the file for sequential files and duplicate records are not appended to KSDS files.

The **AppendToEnd** property and the **OverwriteHostFile** property are mutually exclusive, so it is not possible to enable (set to yes) one of these properties before the opposing property is disabled (set to no). The **AppendToEnd** property takes precedence over the **OverwriteHostFile** property, since **AppendToEnd**defaults to yes and **OverwriteHostFile** defaults to no. Consequently, the order that these properties are set will affect the outcome. For example, the following order will result in the properties being set correctly:

```
FileTransfer.AppendToEnd = eigAnswerNo         // correctly set to no
FileTransfer.OverwriteHostFile = eigAnswerYes  // correctly set to yes
```

In contrast, setting the properties in the improper order will cause the properties to be set incorrectly as follows:

```
FileTransfer.OverwriteHostFile = eigAnswerYes   // remains at no
// AppendToEnd defaults to eigAnswerYes, so this change is illegal
FileTransfer.AppendToEnd = eigAnswerNo          // correctly set to no
```

In this second case, the **OverwriteHostFile** property cannot be set to yes (enabled) until **AppendToEnd** property is set to no (disabled).

Using the Data Descriptions tool, setting the Ascending/Descending option on an OS/390 or MVS/ESA key sequenced file has no effect. Using the Host File Transfer ActiveX Control, data is always uploaded and downloaded in the ascending key order.

If the **Cancel** method is executed while uploading a file with the **AppendToEnd** property set to yes, this will result in no change to the host file. However, if the **Cancel** method is executed while uploading a file with the **OverwriteHostFile** property set to yes, this will result in an empty host file. The **Cancel** method implies the transfer has been stopped and all the files are at their original values but this is not really the case when the **OverwriteHostFile** property is set to yes.

Code Page Support Using Host File Transfer

When using the Host File Transfer ActiveX Control, the Host CCSID (character code set identifier) property should be configured to match the data as represented on the remote host computer. The Host CCSID parameter defaults to EBCDIC U.S./Canada (37) when using the Host File Transfer ActiveX Control.

ISO Code Page Support Using Host File Transfer

Host Integration Server 2000 includes support for some ISO code pages for purposes of ISO-to-UNICODE-to-ANSI, ANSI-to-UNICODE-to-ISO, and ISO-to-UNICODE-to-ISO conversions when using the Host File Transfer ActiveX Control. These ISO code pages can be used when accessing host files containing ISO code pages.

Depending on the version of Windows being used, to support ISO-to-UNICODE-to-ANSI (Windows), ANSI-to-UNCODE-to-ISO, and ISO-to-UNICODE-to-ISO code page conversions, you may need to install the appropriate ISO National Language Support (NLS) file for your locale.

On Windows 2000, the appropriate ISO NLS file for your locale is installed automatically when you install a localized version of Windows 2000.

On Windows NT 4.0, the appropriate ISO NLS file for your locale is installed automatically when you install a localized version of Windows NT or when you install the Windows NT Language Pack on a non-localized version of Windows NT.

On Windows 98 and Windows 95, the appropriate ISO NLS file for your locale is installed automatically when you install a localized version of Windows 98 or Windows 95.

The following table shows the ISO character code set identifiers (CCSIDs) supported by Host File Transfer ActiveX Control in Host Integration Server 2000.

Microsoft Display Name	Microsoft NLS Code Page	IBM CCSID	Comments
ISO 8859-1 Latin 1	28591	819	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-2 Central Europe	28592	912	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 8859-5 Cyrillic	28595	915	On Windows NT 4.0, support for this NLS Code Page is installed using the cyrillic.inf file from the Language Pack.
ISO 8859-6 Arabic	28596	1089	On Windows NT 4.0, support for this NLS Code Page is installed using the arabic.inf file from the Language Pack.
ISO 8859-7 Greek	28597	813	On Windows NT 4.0, support for this NLS Code Page is installed using the greek.inf file from the Language Pack.
ISO 8859-8 Hebrew	28598	916	On Windows NT 4.0, support for this NLS Code Page is installed using the hebrew.inf file from the Language Pack.
ISO 8859-9 Turkish	28599	920	On Windows NT 4.0, support for this NLS Code Page is installed using the european.inf file from the Language Pack.
ISO 6937 Non-Spacing Accent	20269	819	Note that ISO 6937 (CCSID 20269) is not supported by the OLE DB Provider for DB2, but is displayed in the list of configuration options when creating or modifying data sources.
ISO 8859-15 Latin 9 (Euro)	20865	923	NLS Code Page 819 with support for the Euro. On Windows NT 4.0, support for this NLS Code Page is installed using the ibm_euro.inf file from the Language Pack.

The Microsoft Display Name is the name found in the Windows NT Language Pack definitions for these NLS files.

The Microsoft NLS Code Page column represents the code page number that is registered and associated with an ISO-to-UNICODE NLS resource file. The Microsoft NLS number should be set as the Host CCSID when using the Host File Transfer ActiveX Control. When setting the Host CCSID or PC Code Page property, the Microsoft NLS number should be used for this parameter.

The IBM CCSID column represents the CCSID given to the ISO code page in IBM publications. IBM lists their ISO support in publications by referencing the locale name (Bulgaria for ISO8859-5 and 915, for example) rather than simply using ISO 8859-5 Cyrillic as used by Microsoft. The Host File Transfer ActiveX Control does not recognize or display the IBM CCSID values. The Host File Transfer ActiveX Control maps the Microsoft NLS numbers to ISO NLS files which correspond with the appropriate IBM CCSID numbers. The Host File Transfer ActiveX Control passes the corresponding IBM CCSID to the host system at run time even though you configure this property using the Microsoft NLS number.

Note that IBM CCSID 819 is associated with both ISO 8859-1 Latin 1 and ISO 6937 Non-Spacing Accent. It is up to the user to choose the standard ISO 8859-1 Latin 1 code page by selecting NLS code page 28591 or the modified code page ISO 6937 Non-Spacing Accent by selecting NLS code page 20269. Note that ISO 6937 Non-Spacing Accent (CCSID 20269) is not currently supported by the Host File Transfer ActiveX Control.

IBM CCSID 916 (ISO 8859-8) supports Hebrew "visual sort order". IBM CCSID 920 (ISO 8859-8 derivation) supports Hebrew "logical sort order". Although Microsoft supports the logical sort order with NLS 38598, this NLS file is only distributed with Internet Explorer 5 or Windows 2000. The Host File Transfer ActiveX Control has not been tested using the ISO 8859-8 derivation matching IBM CCSID 920 and does not support this configuration.

These are the only ISO pages currently supported in Host Integration Server 2000 and in SNA Server 4.0 with Service Pack 3 or later. Microsoft supports a number of additional ISO pages. IBM also supports additional ISO pages. However, the code pages listed in the table above are the only cases where the Microsoft NLS pages and IBM CCSIDs match.

DBCS Code Page Support Using Host File Transfer

Support for Double-Byte Character String (DBCS) data is limited using the Host File Transfer ActiveX Control. Conversions between DBCS and ANSI code pages are not supported. Conversions between DBCS and ISO code pages are not supported.

Data Conversion Using Host File Transfer

Using the Host File Transfer ActiveX Control, host data is converted to default C data types as defined in ODBC and OLE DB and illustrated in the following table:

Host Data Type (description in HCD file)	**Default C data type**	**Comments**
BINARY		A free form binary data type of specified length. This data type is transferred without being converted.
CHAR	char string[]	A fixed length string. This data type is converted to a DBTYPE_BSTR for use by Host File Transfer ActiveX Control.
DATE	date struct	A 10-byte date string. This data type is converted to a DBTYPE_DATE for use by OLE DB.
DOUBLE	double	An 8-byte double-precision floating point number. This data type is converted to a DBTYPE_R8 for use by OLE DB.
FLOAT	double	An 8-byte double-precision floating point number. This data type is the same as a DOUBLE. This data type is converted to a DBTYPE_R8 for use by OLE DB.
LONG	int	A 4-byte integer ranging in value from -2,147,463,648 to +2,147,483,647. This data type is converted to a DBTYPE_I4 for use by OLE DB.
LONG VARBINARY	char string[]	A varying-length binary string up to 32,740 bytes in length. This data type is converted to a DBTYPE_STR for use by OLE DB.
LONG VARCHAR	char string[]	A varying-length character string up to 32,740 characters in length. This data type is converted to a DBTYPE_STR for use by OLE DB.
PACKED	unsigned char number[]	A packed decimal number. This data type is converted to a DBTYPE_DECIMAL for use by OLE DB
REAL	float	A 4-byte single-precision floating point number. This data type is converted to a DBTYPE_R4 for use by OLE DB.
SHORT	short	A SMALLINT (small integer) is a two-byte integer with a precision of 5 digits ranging from -32,768 to +32,767. This data type is converted to a DBTYPE_I2 for use by OLE DB.
SINGLE	float	A 4-byte single-precision floating point number. This data type is converted to a DBTYPE_R4 for use by OLE DB.
TIME	time struct	An 8-byte time string. This data type is converted to a DBTYPE_TIME for use by OLE DB. When using ActiveX Data Objects to return data from a DB2 TIME data type, ADO returns a DATETIME value.

TIMESTAMP	timestamp struct	A 26-byte string representing the date, time, and microseconds. This data type is converted to a DBTYPE_DBTIMESTAMP for use by OLE DB.
VARBINARY	char string[]	A varying-length binary field. The maximum length of the binary is dependent on the version and the host platform. This data type is transferred without being converted. This data type is converted to a DBTYPE_STR for use by OLE DB.
VARCHAR	char string[]	A varying-length character string. The maximum length of the string is dependent on the version and the host platform. This data type is converted to a DBTYPE_STR for use by OLE DB.
ZONED	unsigned char number[]	A zoned numeric number. This data type is converted to a DBTYPE_NUMERIC for use by OLE DB

Note that the maximum length of fixed-length BINARY, fixed-length CHAR, VARBINARY, and VARCHAR data types is dependent on the version of the host software that is being accessed. For example, the maximum length of the CHAR data type on OS/390 is 254 characters, while the maximum length of this same host data type is 32,765 on OS/400.

Data conversions from a large numeric type to a small numeric type are supported (from DOUBLE to SINGLE and from INT to SMALLINT, for example), however truncation and conversion errors can occur that will not be reported by the Host File Transfer ActiveX Control.

Using the Host File Transfer ActiveX Control, certain conversions of strings from EBCDIC to ASCII and then back to EBCDIC are asymmetric, and can result in strings that are different from the original. The EBCDIC specification contains ordinals for which there is no defined character. The Host File Transfer ActiveX Control translates all such undefined characters to the question mark character ("?"). So when ASCII strings containing these characters are converted back to EBCDIC, these undefined characters will be replaced with question marks. To protect EBCDIC strings containing undefined characters, these fields should be tagged as binary strings and mapped by the application.

The ANSI to EBCDIC character conversions affected include the following:

Character Value (Decimal)	**Character Value (Hexadecimal)**	**ANSI Code Page 1252**	**EBCDIC Character After Conversion to CCSID 37**
128	0x80	Not used	?
130	0x82	Single low quote	?
131	0x83	Latin F with hook	?
132	0x84	Double low quote	?
133	0x85	Ellipsis	?
134	0x86	Dagger	?
135	0x87	Double dagger	?
136	0x88	Per mile	?
137	0x89	S with caron	?
138	0x8A	Left angle	?
139	0x8B	Ligature OE	?
140	0x8C	Not used	?
142	0x8E	Not used	?
145-156	0x91-0x9C		?
158-159	0x9E-0x9F		?

Usernames and Passwords Using Host File Transfer

When connecting to host systems, most users must be authenticated by the remote system by passing a valid User ID and Password.

The AS/400 computer is case sensitive with regard to User ID and Password. The AS/400 only accepts a User ID and Password in uppercase. The Microsoft Host File Transfer ActiveX Control will force the User ID and Password into uppercase when it knows that it is connecting to an AS/400 system.

The mainframe is not case sensitive. This means that on mainframe computers, one can enter the User ID and Password in any case.

Troubleshooting the Host File Transfer ActiveX Control

The Microsoft® Host File Transfer ActiveX Control supplied with Host Integration Server 2000 has the ability to trace DRDA data flows when used over TCP/IP.

This tracing capability is accessible from the SNADB2 Service tracing inside the Trace tool. This facility will show the same data as an APPC trace but without the control indicators (For example, What_Received). Socket errors are traced and the error codes can be looked up in Winsock2.h supplied with the Platform SDK.

The Host File Transfer ActiveX Control can return the following types of errors:

- Errors from the remote hosts
- Microsoft Host File Transfer-specific errors
- Errors from the underlying DDM Application Requester network client

Data Integration Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide describes the objects, methods, properties, controls, and other interfaces that allow you to integrate data into your Host Integration Server application.

This section contains:

OLE DB Object and Interface Support

The OLE DB specification version 2.0 defines a number of objects and interfaces.

The Microsoft® OLE DB Provider for AS/400 and VSAM supports the OLE DB objects and interfaces appropriate for an OLE DB data provider accessing a non-SQL host file system. The following topics provide detailed information on OLE DB support:

- OLE DB Object Support in the OLE DB Provider for AS/400 and VSAM
- OLE DB Interface Support in the OLE DB Provider for AS/400 and VSAM
- OLE DB Property Support in the OLE DB Provider for AS/400 and VSAM

The Microsoft® OLE DB Provider for DB2 supports the OLE DB objects and interfaces appropriate for an OLE DB data provider accessing an SQL database. The following topics provide detailed information on OLE DB support:

- OLE DB Object Support in the OLE DB Provider for DB2
- OLE DB Interface Support in the OLE DB Provider for DB2
- OLE DB Property Support in the OLE DB Provider for DB2

In addition, the following topics provide a comparison of the objects and interfaces supported by the OLE DB Provider for AS/400 and VSAM and the OLE DB Provider for DB2:

- OLE DB Object Support Comparison
- OLE DB Interface Support Comparison

OLE DB Object Support Comparison

The following table compares the OLE DB version 2.0 objects that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft OLE DB Provider for DB2:

OLE DB object	OLE DB Provider for AS/400 and VSAM	OLE DB Provider for DB2
Command	Yes, most interfaces	Yes, most interfaces
CustomErrorObject	No	Yes, all interfaces
DataSource	Yes, most interfaces	Yes, some interfaces
Enumerator	No	No
ErrorObject	Yes, all interfaces	Yes, all interfaces
ErrorRecord	Yes, all interfaces	Yes, all interfaces
Index	Yes, all interfaces	No
MultipleResults	No	No
Rowset	Yes, most interfaces	Yes, some interfaces
Session	Yes, some interfaces	Yes, some interfaces
Transaction	No	Yes, some interfaces
TransactionOptions	No	Yes, all interfaces
View	Yes, all interfaces	No

OLE DB Interface Support Comparison

The following table compares the OLE DB version 2.0 interfaces that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM and the OLE DB Provider for DB2:

Object	Interface	OLE DB Provider for AS/400 and VSAM	OLE DB Provider for DB2
Command	**IAccessor**	Yes	Yes
	IColumnsInfo	Yes	Yes
	IColumnsRowset	No	No
	ICommand	Yes	Yes
	ICommandPersist	No	No
	ICommandPrepare	No	Yes
	ICommandProperties	Yes	Yes
	ICommandText	Yes	Yes
	ICommandWithParameters	No	Yes
	IConvertType	Yes	Yes
	ISupportErrorInfo	Yes	Yes
CustomErrorObject	**IErrorLookup**	No	Yes
	ISQLErrorInfo	No	Yes
DataSource	**IDBAsynchStatus**	No	No
	IConnectionPointContainer	No	No
	IDBCreateSession	Yes	Yes
	IDBDataSourceAdmin	No	No
	IDBInfo	No	Yes
	IDBInitialize	Yes	Yes
	IDBProperties	Yes	Yes
	IPersist	Yes	No
	IPersistFile	Yes	No
	ISupportErrorInfo	Yes	Yes
Enumerator	**IDBInitialize**	No	No
	IDBProperties	No	No
	IParseDisplayName	No	No
	ISourcesRowset	No	No
	ISupportErrorInfo	No	No
ErrorObject	**IErrorRecords**	Yes	Yes
ErrorRecord	**IErrorInfo**	Yes	Yes
Index	**IAccessor**	Yes	No
	IColumnsInfo	Yes	No
	IConvertType	Yes	No
	IRowset	Yes	No
	IRowsetChange	Yes	No
	IRowsetFind	Yes	No
	IRowsetIdentity	Yes	No
	IRowsetIndex	Yes	No
	IRowsetInfo	Yes	No
	IRowsetLocate	Yes	No
	IRowsetRefresh	Yes	No
	IRowsetScroll	Yes	No
	IRowsetUpdate	Yes	No
	IRowsetView	Yes	No
	ISupportErrorInfo	Yes	No
MultipleResults	**IMultipleResults**	No	No
	ISupportErrorInfo	No	No

Rowset	**IAccessor**	Yes	Yes
	IChapteredRowset	Yes	No
	IColumnsInfo	Yes	Yes
	IColumnsRowset	Yes	No
	IConnectionPointContainer	No	No
	IConvertType	Yes	Yes
	IDBAsynchStatus	No	No
	IRowset	Yes	Yes
	IRowsetChange	Yes	Yes
	IRowsetChapterMember	No	No
	IRowsetFind	Yes	No
	IRowsetIdentity	Yes	No
	IRowsetIndex	Yes	No
	IRowsetInfo	Yes	Yes
	IRowsetLocate	Yes	No
	IRowsetRefresh	Yes	No
	IRowsetScroll	No	No
	IRowsetUpdate	Yes	Yes
	IRowsetView	Yes	No
	ISupportErrorInfo	Yes	Yes
Session	**IAlterIndex**	No	No
	IAlterTable	No	No
	IDBCreateCommand	Yes	Yes
	IDBSchemaRowset	Yes	Yes
	IGetDataSource	Yes	Yes
	IIndexDefinition	No	No
	IOpenRowset	Yes	Yes
	ISessionProperties	Yes	Yes
	ISupportErrorInfo	Yes	Yes
	ITableDefinition	No	No
	ITransaction	No	Yes
	ITransactionJoin	No	No
	ITransactionLocal	No	Yes
	ITransactionObject	No	Yes
Transaction	**IConnectionPointContainer**	No	No
	ISupportErrorInfo	No	No
	ITransaction	No	No
TransactionOptions	**ISupportErrorInfo**	No	Yes
	ITransactionOptions	No	Yes
View	**IAccessor**	Yes	No
	IColumnsInfo	Yes	No
	ISupportErrorInfo	Yes	No
	IViewChapter	Yes	No
	IViewFilter	Yes	No
	IViewRowset	Yes	No
	IViewSort	Yes	No

OLE DB Object Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the OLE DB version 2.0 objects that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

OLE DB object	Support
Command	Yes, most interfaces
CustomErrorObject	No
DataSource	Yes, most interfaces
Enumerator	No
ErrorObject	Yes, all interfaces
ErrorRecord	Yes, all interfaces
Index	Yes, all interfaces
MultipleResults	No
Rowset	Yes, most interfaces
Session	Yes, some interfaces
Transaction	No
TransactionOptions	No
View	Yes, all interfaces

OLE DB Interface Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the OLE DB version 2.0 interfaces that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

Object	Interface	Support
Command	**IAccessor**	Yes
	IColumnsInfo	Yes
	IColumnsRowset	No
	ICommand	Yes
	ICommandPersist	No
	ICommandPrepare	No
	ICommandProperties	Yes
	ICommandText	Yes
	ICommandWithParameters	No
	IConvertType	Yes
	ISupportErrorInfo	Yes
CustomErrorObject	**IErrorLookup**	No
	ISQLErrorInfo	No
DataSource	**IDBAsynchStatus**	No
	IDBConnectionPointContainer	No
	IDBCreateSession	Yes
	IDBDataSourceAdmin	No
	IDBInfo	No
	IDBInitialize	Yes
	IDBProperties	Yes
	IPersist	Yes
	IPersistFile	Yes
	ISupportErrorInfo	Yes
Enumerator	**IDBInitialize**	No
	IDBProperties	No
	IParseDisplayName	No
	ISourcesRowset	No
	ISupportErrorInfo	No
ErrorObject	**IErrorRecords**	Yes
ErrorRecord	**IErrorInfo**	Yes
Index	**IAccessor**	Yes
	IColumnsInfo	Yes
	IConvertType	Yes
	IRowset	Yes
	IRowsetChange	Yes
	IRowsetFind	Yes
	IRowsetIdentity	Yes
	IRowsetIndex	Yes
	IRowsetInfo	Yes
	IRowsetLocate	Yes
	IRowsetRefresh	Yes
	IRowsetScroll	Yes
	IRowsetUpdate	Yes
	IRowsetView	Yes
	ISupportErrorInfo	Yes
MultipleResults	**IMultipleResults**	No

	ISupportErrorInfo	No
Rowset	**IAccessor**	Yes
	IChapteredRowset	Yes
	IColumnsInfo	Yes
	IColumnsRowset	Yes
	IConnectionPointContainer	No
	IConvertType	Yes
	IDBAsynchStatus	No
	IRowset	Yes
	IRowsetChange	Yes
	IRowsetChapterMember	No
	IRowsetFind	Yes
	IRowsetIdentity	Yes
	IRowsetIndex	Yes
	IRowsetInfo	Yes
	IRowsetLocate	Yes
	IRowsetRefresh	Yes
	IRowsetScroll	No
	IRowsetUpdate	Yes
	IRowsetView	Yes
	ISupportErrorInfo	Yes
Session	**IAlterIndex**	No
	IAlterTable	No
	IDBCreateCommand	Yes
	IDBSchemaRowset	Yes
	IGetDataSource	Yes
	IIndexDefinition	No
	IOpenRowset	Yes
	ISessionProperties	Yes
	ISupportErrorInfo	Yes
	ITableDefinition	No
	ITransaction	No
	ITransactionJoin	No
	ITransactionLocal	No
	ITransactionObject	No
Transaction	**IConnectionPointContainer**	No
	ISupportErrorInfo	No
	ITransaction	No
TransactionOptions	**ISupportErrorInfo**	No
	ITransactionOptions	No
View	**IAccessor**	Yes
	IColumnsInfo	Yes
	ISupportErrorInfo	Yes
	IViewChapter	Yes
	IViewFilter	Yes
	IViewRowset	Yes
	IViewSort	Yes

OLE DB Provider for AS/400 and VSAM Command Object

The **Command** object is created by an OLE DB consumer, or by a service provider on behalf of a consumer. A **Command** object is used to execute a DDM-specific command on a remote DDM server. The **Command** object currently supports executing Command Language commands on AS/400 DDM servers.

It is important not to confuse a command, which is an OLE COM object, and its command text, which is a string. Commands are generally used for data definition, such as creating a table or granting privileges, and data manipulation, such as updating or deleting rows. A special case of data manipulation using the **Command** object is opening a rowset (a table).

Before a consumer can use a command, it must determine if commands are supported. To do this, the consumer calls **QueryInterface** for **IDBCreateCommand** on a session. If this interface is exposed, the provider supports commands. To create a command, the consumer then calls **IDBCreateCommand::CreateCommand** on the session. A single session can be used to create multiple commands.

When the command is first created, it does not contain a command text. The consumer sets the command text with **ICommandText::SetCommandText**. Because the text command syntax is provider-specific, the consumer passes the GUID of the syntax to use. For use with Microsoft® OLE DB Provider for AS/400 and VSAM, the GUID is DBGUID_DBSQL. Please note that under the OLE DB Provider for AS/400 and VSAM, this GUID does not signify that the text command is a superset of ANSI SQL. The level at which the provider supports ANSI SQL is specified by the DBPROP_SQLSUPPORT property. This property is a bit mask specifying the level of support for SQL. The OLE DB Provider for AS/400 and VSAM sets this property to DBPROPVAL_SQL_NONE, indicating that SQL is not supported.

The syntax supported by the OLE DB Provider for AS/400 and VSAM for command text is as follows:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

The syntax supported by the OLE DB Provider for AS/400 and VSAM to open a rowset (table) using command text is as follows:

```
EXEC OPEN FileName
```

where *FileName* represents one of the following host file naming conventions:

Host file type	File naming convention
VSAM Data Sets	DATASETNAME.FILENAME
Partitioned Data Sets	DATASETNAME.FILENAME(MEMBER)
OS/400 Files	LIBRARY/FILE
OS/400 Files	LIBRARY/FILENAME
OS/400 File Members	LIBRARY/FILE(MEMBER)
OS/400 File Members	LIBRARY.FILENAME(MEMBER)

Note that if a member of a library contains a dot in the member name, the member name must be surrounded by double quotes. For example, if the member name is NAMES.DAT, the proper syntax used to open a rowset using command text is as follows:

```
EXEC OPEN LIBRARY/FILE("NAMES.DAT")
```

To execute the command, the consumer calls **ICommand::Execute**. If the command text specifies the command to open a rowset, (an **EXEC OPEN** command), **Execute** instantiates the rowset and returns an interface pointer to it.

The following interfaces of the **Command** object are supported by the current version of the OLE DB Provider for AS/400 and VSAM:

- **IAccessor**
- **IColumnsInfo**
- **ICommand**

- **ICommandProperties**
- **ICommandText**
- **IConvertType**
- **ISupportErrorInfo**

OLE DB Provider for AS/400 and VSAM DataSource Object

The **DataSource** object is created by an OLE DB consumer. The **DataSource** object contains the knowledge and ability to connect to an IBM mainframe or AS/400 over APPC and LU6.2 (through Microsoft® Host Integration Server 2000) or over TCP/IP. The **DataSource** object is used to create one or more **Session** objects.

The following interfaces of the **DataSource** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

- **IDBCreateSession**
- **IDBInitialize**
- **IDBProperties**
- **IPersist**
- **IPersistFile**
- **ISupportErrorInfo**

OLE DB Provider for AS/400 and VSAM ErrorObject Object

The **ErrorObject** object is created by any interface on any SNA OLE DB object. The **ErrorObject** object is used to retrieve additional information when an error occurs.

The following interfaces of the **ErrorObject** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

- **IErrorRecords**

The IErrorRecords interface returns **ErrorRecord** objects with detailed information on the error that occurred.

OLE DB Provider for AS/400 and VSAM ErrorRecord Object

The **ErrorRecord** object is created by calling the IErrorRecord interface on the **ErrorObject** object. An **ErrorObject** is created on any interface on any SNA OLE DB object when an error occurs. The **ErrorRecord** object is used to retrieve additional information when an error occurs.

The following interfaces of the **ErrorRecord** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

- **IErrorInfo**

OLE DB interface methods return error information in two ways. The error code returned by an interface method, known as the *return code*, indicates the overall success or failure of a method. Error records provide detailed information about the error, such as a text description of the error, the GUID of the interface that defined the error, and provider-specific error information. Error objects in OLE DB are an extension of the error objects in Automation, they use many of the same mechanisms, and can be used as Automation error objects.

OLE DB error return codes are of type HRESULT. There are two general classes of return codes: success and warning codes, and error codes.

Success and warning codes begin with S_ or DB_S_ and indicate that the method successfully completed. The standard OLE DB error codes are defined in the OLEDBERR.H include file.

If the return code is other than S_OK or S_FALSE, it is likely that an error occurred from which the method was able to recover. For example, **IRowset::GetNextRows** returns DB_S_ENDOFROWSET when it is unable to return the requested number of rows due to reaching the end of the rowset. If a single warning condition occurs, the method returns the code for that condition. If multiple warning conditions occur, the method describes the hierarchy of warning return codes indicating which warning code should be returned when given a choice between multiple warning return codes.

Error codes begin with E_ or DB_E_ and indicate that the method failed completely and was unable to do any useful work. For example, **GetNextRows** returns E_INVALIDARG when the pointer in which it is to return a pointer to an array of row handles (*prghRows*) is null. An exception to this is that some of the methods that return DB_E_ERRORSOCCURRED allocate memory in which to return additional information about these errors. Consumers must free this memory. For information about which methods allocate memory in this case, see the methods that return DB_E_ERRORSOCCURRED. Although error codes can indicate run-time errors, such as running out of memory, they generally indicate programming errors. If multiple errors occur, the code that is returned is provider-specific. If both errors and warnings occur, the method fails and returns an error code.

All methods can return S_OK, E_FAIL, and E_OUTOFMEMORY. The E_OUTOFMEMORY code applies only to those methods which allocate memory that is returned to the consumer. In some cases, the E_OUTOFMEMORY code might be eliminated by calling the method requesting fewer returned values, such as fewer rows from **GetNextRows**.

OLE DB Provider for AS/400 and VSAM Index Object

An OLE DB index, also known as an *index rowset*, is a rowset built over an index in a data source. It is generally used in conjunction with a rowset built over a base table in the same data source. Each row of the index rowset contains a bookmark that points to a row in the base-table rowset. Thus, an OLE DB consumer can traverse the index rowset and use it to access rows in the base-table rowset.

Indexes are created using the Index interfaces of the **Rowset** object. Index rowsets allow an application to read records efficiently by means of a key.

The following Index interfaces of the **Rowset** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM when applied to AS/400 keyed physical files, AS/400 logical files, VSAM KSDS files with unique keys, and VSAM RRDS files with unique keys.

- **IAccessor**
- **IColumnsInfo**
- **IConvertType**
- **IRowset**
- **IRowsetChange**
- **IRowsetFind**
- **IRowsetIdentity**
- **IRowsetIndex**
- **IRowsetInfo**
- **IRowsetLocate**
- **IRowsetRefresh**
- **IRowsetScroll**
- **IRowsetUpdate**
- **IRowsetView**
- **ISupportErrorInfo**

The OLE DB Provider for AS/400 and VSAM supports integrated indexes using the **IRowsetIndex** interface based on the underlying rowset. For more information on indexes, see Chapter 8, "Indexes" and Chapter 16, "Integrated Indexes" in the *OLE DB Programmer's Reference*.

OLE DB Provider for AS/400 and VSAM Rowset Object

Rowset objects are created by **Session** objects. The **Rowset** object exposes data in tabular format.

The following interfaces of the **Rowset** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

- **IAccessor**
- **IChapteredRowset**
- **IColumnsInfo**
- **IColumnsRowset**
- **IConvertType**
- **IRowset**
- **IRowsetChange**
- **IRowsetFind**
- **IRowsetIdentity**
- **IRowsetIndex**
- **IRowsetInfo**
- **IRowsetLocate**
- **IRowsetRefresh**
- **IRowsetUpdate**
- **IRowsetView**
- **ISupportErrorInfo**

> **Note** The IRowsetFind interface is only supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM when applied to AS/400 keyed physical files, AS/400 logical files, VSAM KSDS files with unique keys, and VSAM RRDS files with unique keys.

OLE DB Provider for AS/400 and VSAM Session Object

The **Session** object is created by a **DataSource** object. The **Session** object is used to create one or more **Rowset** objects.

The following interfaces of the **Session** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

- **IDBCreateCommand**
- **IDBSchemaRowset**
- **IGetDataSource**
- **IOpenRowset**
- **ISessionProperties**
- **ISupportErrorInfo**

Consumers can get information about a data store without knowing its structure by using the **IDBSchemaRowset** methods. The methods on this interface can be used to retrieve advanced schema information. The OLE DB Provider for AS/400 and VSAM represents organizes this information into a set of schemas that contain tables for each schema. These schema rowsets are identifed by GUIDs.

The following schema rowset GUIDs are supported by the OLE DB Provider for AS/400 and VSAM:

- DBSCHEMA_COLUMNS
- DBSCHEMA_INDEXES
- DBSCHEMA_PROVIDER_TYPES
- DBSCHEMA_TABLES

The following table lists these GUIDs and the columns for which restrictions can be specified on the schema rowset when using the OLE DB Provider for AS/400 and VSAM. The number of restriction columns for each schema rowset are defined as constants prefixed with CRESTRICTIONS_ in the OLEDB header files. Restriction values are treated as literals rather than as search patterns. For example, the restriction value "A_C" matches "A_C" but not "ABC".

GUID	Number of Restrictions	Restriction Columns
DBSCHEMA_COLUMNS	4	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
DBSCHEMA_INDEXES	5	TABLE_CATALOG TABLE_SCHEMA INDEX_NAME TYPE TABLE_NAME
DBSCHEMA_PROVIDER_TYPES	2	DATA_TYPE BEST_MATCH
DBSCHEMA_TABLES	4	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE

The **IDBSchemaRowset** interface allows an application to pass at run time the target library of a Partioned Data Set (PDS/PDSE), a dataset, or a member name when using the IDBSchemaRowset:GetSchemas function to retrieve the schema.

This following sample illustrates using a target library to retrieve a table schema:

```
hr = pIDBSchemaRowset->GetRowset(
        NULL,                                   // punkOuter
        DBSCHEMA_TABLES,     // schema IID
         2L,                                       // # of restrictions
         rgRestrictions,                    // array of restrictions
         IID_IRowset,                       // rowset interface
         0L,                                        // # of properties
         NULL,                                   // properties
         (IUnknown**)&pIRowset);  // rowset pointer
```

The variable *rgRestrictions* is an array containing two restriction values. The first array entry is VT_EMPTY and the second array entry is the target library name.

OLE DB Provider for AS/400 and VSAM View Object

The **View** object is created on a **Rowset** object. The **View** object is used to expose simple operations, such as sorting and filtering a rowset by applying a view. Views can be applied when opening a **Rowset** object or applied to an existing **Rowset** object.

The following interfaces of the **View** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM when applied to AS/400 keyed physical files, AS/400 logical files, VSAM KSDS files with unique keys, and VSAM RRDS files with unique keys.

- **IAccessor**
- **IColumnsInfo**
- **ISupportErrorInfo**
- **IViewChapter**
- **IViewFilter**
- **IViewRowset**
- **IViewSort**

OLE DB Property Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the provider-specific OLE DB version 2.0 properties in the SNAOLEDB_DBPROPSET_DBINIT property set that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM.

OLE DB Property ID	Description
DBPROP_SNAOLEDB_APPCMODE	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_SNAOLEDB_NETTYPE), this property is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This VT_BSTR type property normally defaults to QPCSUPP.
DBPROP_SNAOLEDB_BINASCHAR	This property indicates whether to process binary fields (CCSID of 65535) as character data type fields on a per data source basis. The host CCSID and PC Code Page values are required input parameters when this parameter is true. This VT_BOOL type property defaults to VARIANT_FALSE, don't process binary fields as character fields.
DBPROP_SNAOLEDB_HCDPATH	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This VT_BSTR type property is required when connecting to mainframe systems and is optional when connecting to OS/400.
DBPROP_SNAOLEDB_HOSTCCSID	The character code set identifier (CCSID) matching the data as represented on the host. This property is required when processing binary data as character data. Unless the DBPROP_SNAOLEDB_BINASCHAR property is set to true, character data is converted based on the host column CCSID and default ANSI code page. This VT_I4 property defaults to U.S./Canada (37).
DBPROP_SNAOLEDB_LIBRARY	The default AS/400 library to be accessed. This VT_BSTR property is not required for mainframe access and is optional when connecting to AS/400 files.
DBPROP_SNAOLEDB_LOCALLU	When LU 6.2 (SNA) is selected for the Network Transport Library, this property is the name of the local LU alias configured in the SNA server.
DBPROP_SNAOLEDB_NETADDRESS	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target host computer. This parameter indicates the IP address or TCP/IP host name alias associated with the DDM server on the host. The network address is required when connecting via TCP/IP.

DBPROP_SNAOLEDB_NETPORT	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target DDM service access port when connecting via TCP/IP. This parameter represents the TCP/IP port used for communication with the DDM service on the host. The default value for the VT_BSTR type property is 446.
DBPROP_SNAOLEDB_NETTYPE	This property which represents the dynamic link library used for transport designates whether the provider connects via SNA LU 6.2 or TCP/IP for network communication. The possible values for this parameter are TCP/IP or SNA. If TCPIP is selected, then values for Network Address (DBPROP_SNAOLEDB_NETADDRESS) and Network Port (DBPROP_SNAOLEDB_NETPORT) are required. TCP/IP connectivity to the mainframe is not supported by the OLE DB Provider for AS/400 and VSAM. If SNA is selected, then values for APPC Local LU Alias (DBPROP_SNAOLEDB_LOCALLU, APPC Mode Name (DBPROP_SNAOLEDB_APPCMODE), and APPC Remote LU Alias (DBPROP_SNAOLEDB_REMOTELU) are required. This value for this VT_BSTR property defaults to SNA.
DBPROP_SNAOLEDB_PCCODEPAGE	The PC Code Page property ID indicates the code page to be used on the PC for character code conversion. This property is required when processing binary data as character data. Unless DBPROP_SNAOLEDB_BINASCHAR is set to true, character data is converted based on the default ANSI code page configured in Windows. If this parameter is set to Binary or 65535, then no character code conversions will take place. The default value for this VT_I4 type property is 1252 (Latin-1).
DBPROP_SNAOLEDB_REMOTELU	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_SNAOLEDB_NETTYPE), this property ID is the name of the remote LU alias configured in the SNA server.
DBPROP_SNAOLEDB_REPAIRKEY	This property ID provides for repair of invalid key offsets received from OS/400 when keys have been defined using the DDS "RENAME" clause. This parameter indicates whether the OLE DB provider should repair any host key values set in the registry. This VT_BOOL type property defaults to VARIANT_FALSE.
DBPROP_SNAOLEDB_STRICTVAL	This property indicates whether strict validation should be used. This VT_BOOL type property defaults to VARIANT_FALSE.

OLE DB Object Support in the OLE DB Provider for DB2

The following table summarizes the OLE DB version 2.0 objects that are supported by the current version of the Microsoft® OLE DB Provider for DB2:

OLE DB object	Support
Command	Yes, most interfaces
CustomErrorObject	Yes, all interfaces
DataSource	Yes, some interfaces
Enumerator	No
ErrorObject	Yes, all interfaces
ErrorRecord	Yes, all interfaces
Index	No
MultipleResults	No
Rowset	Yes, some interfaces
Session	Yes, some interfaces
Transaction	Yes, some interfaces
TransactionOptions	Yes, all interfaces
View	No

OLE DB Interface Support in the OLE DB Provider for DB2

The following table summarizes the OLE DB version 2.0 interfaces that are supported by the current version of the Microsoft® OLE DB Provider for DB2:

Object	Interface	Support
Command	**IAccessor**	Yes
	IColumnsInfo	Yes
	IColumnsRowset	No
	ICommand	Yes
	ICommandPersist	No
	ICommandPrepare	Yes
	ICommandProperties	Yes
	ICommandText	Yes
	ICommandWithParameters	Yes
	IConvertType	Yes
	ISupportErrorInfo	Yes
CustomErrorObject	**IErrorLookup**	Yes
	ISQLErrorInfo	Yes
DataSource	**IDBAsynchStatus**	No
	IConnectionPointContainer	No
	IDBCreateSession	Yes
	IDBDataSourceAdmin	No
	IDBInfo	Yes
	IDBInitialize	Yes
	IDBProperties	Yes
	IPersist	Yes
	IPersistFile	Yes
	ISupportErrorInfo	Yes
Enumerator	**IDBInitialize**	No
	IDBProperties	No
	IParseDisplayName	No
	ISourcesRowset	No
	ISupportErrorInfo	No
ErrorObject	**IErrorRecords**	Yes
ErrorRecord	**IErrorInfo**	Yes
Index	**IAccessor**	No
	IColumnsInfo	No
	IConvertType	No
	IRowset	No
	IRowsetChange	No
	IRowsetFind	No
	IRowsetIdentity	No
	IRowsetIndex	No
	IRowsetInfo	No
	IRowsetLocate	No
	IRowsetRefresh	No
	IRowsetScroll	No
	IRowsetUpdate	No
	IRowsetView	No
	ISupportErrorInfo	No
MultipleResults	**IMultipleResults**	No
	ISupportErrorInfo	No
Rowset	**IAccessor**	Yes

	IChapteredRowset	No
	IColumnsInfo	Yes
	IColumnsRowset	No
	IConnectionPointContainer	No
	IConvertType	Yes
	IDBAsynchStatus	No
	IRowset	Yes
	IRowsetChange	Yes
	IRowsetChapterMember	No
	IRowsetFind	No
	IRowsetIdentity	No
	IRowsetIndex	No
	IRowsetInfo	Yes
	IRowsetLocate	No
	IRowsetRefresh	No
	IRowsetScroll	No
	IRowsetUpdate	Yes
	IRowsetView	No
	ISupportErrorInfo	Yes
Session	**IAlterIndex**	No
	IAlterTable	No
	IDBCreateCommand	Yes
	IDBSchemaRowset	Yes
	IGetDataSource	Yes
	IIndexDefinition	No
	IOpenRowset	Yes
	ISessionProperties	Yes
	ISupportErrorInfo	Yes
	ITableDefinition	No
	ITransaction	Yes
	ITransactionJoin	No
	ITransactionLocal	Yes
	ITransactionObject	Yes
Transaction	**IConnectionPointContainer**	No
	ISupportErrorInfo	Yes
	ITransaction	Yes
TransactionOptions	**ISupportErrorInfo**	Yes
	ITransactionOptions	Yes
View	**IAccessor**	No
	IColumnsInfo	No
	ISupportErrorInfo	No
	IViewChapter	No
	IViewFilter	No
	IViewRowset	No
	IViewSort	No

OLE DB Provider for DB2 Command Object

The **Command** object is created by an OLE DB consumer, or by a service provider on behalf of a consumer. A **Command** object is used to execute a DDM-specific command on a remote DDM server. The **Command** object currently supports executing Command Language commands on AS/400 DDM servers.

It is important not to confuse a command, which is an OLE COM object, and its command text, which is a string. Commands are generally used for data definition, such as creating a table or granting privileges, and data manipulation, such as updating or deleting rows. A special case of data manipulation using the **Command** object is the creation of rowsets based on DB2 tables. When using the command text with DB2/400 on the AS/400, table names specified in a command are by default passed as uppercase. If a table name uses mixed case, then the table name must be passed in a quoted string.

Before a consumer can use a command, it must determine if commands are supported. To do this, the consumer calls **QueryInterface** for **IDBCreateCommand** on a session. If this interface is exposed, the provider supports commands. To create a command, the consumer then calls **IDBCreateCommand::CreateCommand** on the session. A single session can be used to create multiple commands.

When the command is first created, it does not contain a command text. The consumer sets the command text with **ICommandText::SetCommandText**. Because the text command syntax is provider-specific, the consumer passes the GUID of the syntax to use. For use with Microsoft® OLE DB Provider for DB2, the GUID is DBGUID_DBSQL. Please note that under the OLE DB Provider for DB2, this GUID signifies that the text command is a superset of ANSI SQL. The level at which the provider supports ANSI SQL is specified by the DBPROP_SQLSUPPORT property. This property is a bit mask specifying the level of support for SQL.

The syntax supported by the OLE DB Provider for DB2 for command text is as Entry-Level ANSI SQL 92 (with some exceptions based on the DB2 server host platform).

Legal SQL commands are documented in the following publications published by IBM:

- AS/400 Advanced Series: DB2 for AS/400 SQL Reference Version 4 (Document Number SC41-5612-00)
- DB2 for OS/390 Version 5: SQL Reference (Document Number SC26-8966)

To execute the command, the consumer calls **ICommand::Execute**. If the command text specifies the command to open a rowset, **Execute** instantiates the rowset and returns an interface pointer to it.

The following interfaces of the **Command** object are supported by the current version of the OLE DB Provider for DB2.

- **IAccessor**
- **IColumnsInfo**
- **ICommand**
- **ICommandPrepare**
- **ICommandProperties**
- **ICommandText**
- **ICommandWithParameters**
- **IConvertType**
- **ISupportErrorInfo**

When using the **ICommand** object, the Microsoft OLE DB Provider for DB2 cannot derive parameter type information from the data store. The OLE DB client application must supply the native parameter type information through **ICommandWithParameters::SetParameterInfo** function. The OLE DB provider uses the type information specified by **SetParameterInfo** to determine how to convert parameter data from the type supplied by the consumer (as indicated by the wType value in the binding structure) to the native type used by the data store. When the consumer specifies a data type with known precision, scale, and size values, any information supplied by the consumer for precision, scale, or size is ignored by the OLE DB Provider for DB2.

The information that the consumer supplies must be correct and must be supplied for all parameters. The OLE DB Provider for DB2 cannot verify the supplied information against the parameter metadata, although the OLE DB provider can determine that the specified values are legal values for the provider. The result of executing a command using incorrect parameter information or passing parameter information for the wrong number of parameters is undefined. For example, if the parameter type is LONG and the consumer specifies a type indicator of DBTYPE_STR in **ICommandWithParameters::SetParameterInfo**, the OLE DB Provider for DB2 converts the data to a string before sending it to the data store. Because the data store is unepecting a LONG, this will likely result in an error.

OLE DB Provider for DB2 CustomErrorObject Object

The **CustomErrorObject** object is created by a **Command** object when a command error occurs. The **CustomErrorObject** object is used to retrieve additional information when an error occurs.

The following interfaces of the **CustomErrorObject** object are supported by the current version of the Microsoft® OLE DB Provider for DB2:

- **IErrorLookup**
- **ISQLErrorInfo**

OLE DB Provider for DB2 DataSource Object

The **DataSource** object is created by an OLE DB consumer. The **DataSource** object contains the knowledge and ability to connect to DB2 over APPC and LU6.2 (through Microsoft® Host Integration Server 2000) or over TCP/IP. The **DataSource** object is used to create one or more **Session** objects.

The following interfaces of the **DataSource** object are supported by the current version of the Microsoft® OLE DB Provider for DB2:

- **IDBCreateSession**
- **IDBInfo**
- **IDBInitialize**
- **IDBProperties**
- **IPersist**
- **IPersistFile**
- **ISupportErrorInfo**

OLE DB Provider for DB2 ErrorObject Object

The **ErrorObject** object is created by any interface on any DB2 OLE DB object. The **ErrorObject** object is used to retrieve additional information when an error occurs.

The following interfaces of the **ErrorObject** object are supported by the current version of the Microsoft® OLE DB Provider for DB2:

- **IErrorRecords**

The IErrorRecords interface returns **ErrorRecord** objects with detailed information on the error that occurred.

OLE DB Provider for DB2 ErrorRecord Object

The **ErrorRecord** object is created by calling the IErrorRecord interface on the **ErrorObject** object. An **ErrorObject** is created on any interface on any SNA OLE DB object when an error occurs. The **ErrorRecord** object is used to retrieve additional information when an error occurs.

The following interface of the **ErrorRecord** object is supported by the current version of the Microsoft® OLE DB Provider for DB2:

- **IErrorInfo**

OLE DB interface methods return error information in two ways. The error code returned by an interface method, known as the *return code*, indicates the overall success or failure of a method. Error records provide detailed information about the error, such as a text description of the error, the GUID of the interface that defined the error, and provider-specific error information. Error objects in OLE DB are an extension of the error objects in Automation, they use many of the same mechanisms, and can be used as Automation error objects.

OLE DB error return codes are of type HRESULT. There are two general classes of return codes: success and warning codes, and error codes.

Success and warning codes begin with S_ or DB_S_ and indicate that the method successfully completed. The standard OLE DB error codes are defined in the OLEDBERR.H include file.

If the return code is other than S_OK or S_FALSE, it is likely that an error occurred from which the method was able to recover. For example, **IRowset::GetNextRows** returns DB_S_ENDOFROWSET when it is unable to return the requested number of rows due to reaching the end of the rowset. If a single warning condition occurs, the method returns the code for that condition. If multiple warning conditions occur, the method describes the hierarchy of warning return codes, indicating which warning code should be returned when given a choice between multiple warning return codes.

Error codes begin with E_ or DB_E_ and indicate that the method failed completely and was unable to do any useful work. For example, **GetNextRows** returns E_INVALIDARG when a null pointer in which the OLE DB Provider to return a pointer to an array of row handles (*prghRows*). An exception to this is that some of the methods that return DB_E_ERRORSOCCURRED allocate memory to return additional information about these errors. Consumers must free this memory. For information about which methods allocate memory in this case, see the methods that return DB_E_ERRORSOCCURRED. Although error codes can indicate run-time errors, such as running out of memory, they generally indicate programming errors. If multiple errors occur, the code that is returned is provider-specific. If both errors and warnings occur, the method fails and returns an error code.

All methods can return S_OK, E_FAIL, and E_OUTOFMEMORY. The E_OUTOFMEMORY code applies only to those methods which allocate memory that is returned to the consumer. In some cases, the E_OUTOFMEMORY code might be eliminated by calling the method requesting fewer returned values, such as fewer rows from **GetNextRows**.

OLE DB Provider for DB2 Rowset Object

Rowset objects are created by **Session** objects. The **Rowset** object exposes data in tabular format.

The following interfaces of the **Rowset** object are supported by the current version of the Microsoft® OLE DB Provider for DB2.

- **IAccessor**
- **IColumnsInfo**
- **IConvertType**
- **IRowset**
- **IRowsetChange**
- **IRowsetInfo**
- **IRowsetUpdate**
- ISupportErrorInfo

OLE DB Provider for DB2 Session Object

The **Session** object is created by a **DataSource** object. The **Session** object is used to create one or more **Rowset** objects.

The following interfaces of the **Session** object are supported by the current version of the Microsoft® OLE DB Provider for DB2.

- **IDBCreateCommand**
- **IDBSchemaRowset**
- **IGetDataSource**
- **IOpenRowset**
- **ISessionProperties**
- **ISupportErrorInfo**
- **ITransaction**
- **ITransactionLocal**
- **ITransactionObject**

Consumers can get information about a data store without knowing its structure by using the **IDBSchemaRowset** methods. The methods on this interface can be used to retrieve advanced schema information. The OLE DB Provider for DB2 represents each DB2 database server into a set of schemas that contain tables for each schema. These schema rowsets are identifed by GUIDs.

The following schema rowset GUIDs are supported by the OLE DB Provider for DB2:

- DBSCHEMA_COLUMNS
- DBSCHEMA_INDEXES
- DBSCHEMA_PRIMARY_KEYS
- DBSCHEMA_PROCEDURES
- DBSCHEMA_PROCEDURE_PARAMETERS
- DBSCHEMA_PROVIDER_TYPES
- DBSCHEMA_TABLES

The following table lists these GUIDs and the columns for which restrictions can be specified on the schema rowset when using the OLE DB Provider for DB2. The number of restriction columns for each schema rowset are defined as constants prefixed with CRESTRICTIONS_ in the OLEDB header files. Restriction values are treated as literals rather than as search patterns. For example, the restriction value "A_C" matches "A_C" but not "ABC".

GUID	Number of Restrictions	Restriction Columns
DBSCHEMA_COLUMNS	4	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
DBSCHEMA_INDEXES	4	TABLE_CATALOG TABLE_SCHEMA INDEX_NAME TABLE_NAME
DBSCHEMA_PRIMARY_KEYS	3	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
DBSCHEMA_PROCEDURES	4	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PROCEDURE_TYPE
DBSCHEMA_PROCEDURE_PARAMETERS	4	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PARAMETER_NAME
DBSCHEMA_PROVIDER_TYPES	2	DATA_TYPE BEST_MATCH

DBSCHEMA_TABLES	4	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE

Note that the TYPE restriction on the DBSCHEMA_INDEXES GUID is not supported by the OLE DB Provider for DB2.

The PROCEDURE_SCHEMA restriction on the DBSCHEMA_PROCEDURE GUID and the DBSCHEMA_PROCEDURE_PARAMETERS GUID is not supported when connecting to DB/2 on OS/390 platforms.

OLE DB Provider for DB2 Transaction Object

The **Transaction** object is created by a **Session** object. The **Transaction** object is used to manage transactions on one or more **Rowset** objects.

The following interfaces of the **Transaction** object are supported by the current version of the Microsoft® OLE DB Provider for DB2.

- **ISupportErrorInfo**
- **ITransaction**

The current implementation of the OLE DB Provider for DB2 services all OLE DB Session, Command, and Rowset objects present in a given instance of the DataSource object through a single APPC conversation or TCP/IP connection. One implication of this design is that if two Rowset objects, each created from a different OLE DB Session object, use explicit commitment control through the **ITransaction** interface, they will interfere with each other. When a Commit or Abort for one instance is invoked, all work for the DataSource object will be either committed or aborted. This may yield undesirable results. The work around to this problem is to instantiate two instances of the DataSource object.

OLE DB Property Support in the OLE DB Provider for DB2

The OLE DB Provider for DB2 included with Host Integration Server 2000 supports a different set of provider-specific properties than the earlier OLE DB Provider for DB2 supplied with SNA Server 4.0. The sections below provide information on provider-specific and standard OLE DB properties supported by the current and the earlier OLE DB provider.

This section contains:

OLE DB Provider-Specific Property Support in the OLE DB Provider for DB2

The following table summarizes the provider-specific OLE DB version 2.0 properties in the DB2OLEDB_DBPROPSET_DBINIT property set that are supported by the version of the Microsoft® OLE DB Provider for DB2 included with Host Integration Server 2000:

OLE DB Property ID	Description
DBPROP_DB2OLEDB_APPCMODE	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_DB2OLEDB_NETTYPE), this property is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This VT_BSTR type property normally defaults to QPCSUPP.
DBPROP_DB2OLEDB_BINASCHAR	This property indicates whether to process binary fields (CCSID of 65535) as character data type fields on a per data source basis. The host CCSID and PC Code Page values are required input parameters when this parameter is true. This VT_BOOL type property defaults to VARIANT_FALSE, don't process binary fields as character fields.
DBPROP_DB2OLEDB_CATALOGCOL	The name of the collection where the OLE DB Provider for DB2 looks for catalog information. This is the default schema, the "SCHEMA" name for the target collection of tables and views. This property is the Data Schema value when configuring data sources. The OLE DB Provider for DB2 uses this default schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection. For DB2, the default schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the default schema is the target COLLECTION name. For DB2 Universal Database (UDB), the default schema is the SCHEMA name. If the user does not provide a VT_BSTR value for DBPROP_DB2OLEDB_CATALOGCOL, then the OLE DB Provider uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, these values for the default schema are inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DBPROP_DB2OLEDB_HOSTCCSID	The character code set identifier (CCSID) matching the data as represented on the host. This property is required when processing binary data as character data. Unless the DBPROP_DB2OLEDB_BINASCHAR property ID is set to true, character data is converted based on the host column CCSID and default ANSI code page. This VT_I4 type property defaults to 37 (U.S./Canada).
DBPROP_DB2OLEDB_LOCALLU	When LU 6.2 (SNA) is selected for the Network Transport Library, this property is the name of the local LU alias configured in the SNA server. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_NETADDRESS	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target host computer. This parameter indicates the IP address or TCP/IP host name alias associated with the DDM server on the host. The network address is required when connecting via TCP/IP. This VT_BSTR type property defaults to SNA.

DBPROP_DB2OLEDB_NETPORT	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target DDM service access port when connecting via TCP/IP. This parameter represents the TCP/IP port used for communication with the DDM service on the host. This VT_BSTR type property defaults to 446.
DBPROP_DB2OLEDB_NETTYPE	This property which represents the dynamic link library used for transport designates whether the provider connects via SNA LU 6.2 or TCP/IP for network communication. The possible values for this parameter are TCP/IP or SNA. If TCPIP is selected, then values for Network Address (DBPROP_DB2OLEDB_NETADDRESS) and Network Port (DBPROP_DB2OLEDB_NETPORT) are required. If SNA is selected, then values for APPC Local LU Alias (DBPROP_DB2OLEDB_LOCALLU, APPC Mode Name (DBPROP_DB2OLEDB_APPCMODE), and APPC Remote LU Alias (DBPROP_DB2OLEDB_REMOTELU) are required. This VT_BSTR type property defaults to SNA.
DBPROP_DB2OLEDB_PACKAGECOL	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be the same as the Default Schema (DBPROP_DB2OLEDB_DEFAULTSCH). The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The OLE DB Provider for DB2 will create packages dynamically in the location to which the user points using the this property ID. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_PCCODEPAGE	The PC Code Page property ID indicates the code page to be used on the PC for character code conversion. This property is required when processing binary data as character data. Unless DBPROP_DB2OLEDB_BINASCHAR is set to true, character data is converted based on the default ANSI code page configured in Windows. If this parameter is set to Binary or 65535, then no character code conversions will take place. This VT_I4 type property defaults to 1252 (Latin 1).
DBPROP_DB2OLEDB_REMOTELU	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_DB2OLEDB_NETTYPE), this property is the name of the remote LU alias configured in the SNA server. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_TPNAME	This property represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, this property is set to 0X07F9F9F9. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_UNITSOFWORK	This property indicates whether two-phase commit (distributed unit of work) used for transactions is supported for this data source. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. The following values for this property are supported by the OLE DB Provider for DB2: RUW (remote unit of work) DUW (distributed unit of work) This VT_BSTR type property has a default value of RUW. Distributed unit of work (two-phase commit) works only with DB2 for OS/390 v5R1 or later. This option also requires that the SNA LU 6.2 service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

The following table summarizes the provider-specific OLE DB version 2.0 properties in the DB2OLEDB_DBPROPSET_DBINIT property set that are supported by the version of the Microsoft® OLE DB Provider for DB2 included with SNA Server 4.0:

OLE DB Property ID	**Description**

DBPROP_DB2OLEDB_APPCMODE	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_DB2OLEDB_NETTYPE), this property is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This VT_BSTR type property normally defaults to QPCSUPP.
DBPROP_DB2OLEDB_BINASCHAR	This property indicates whether to process binary fields (CCSID of 65535) as character data type fields on a per data source basis. The host CCSID and PC Code Page values are required input parameters when this parameter is true. This VT_BOOL type property defaults to VARIANT_FALSE, don't process binary fields as character fields.
DBPROP_DB2OLEDB_BINDTYPE	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The NORM package binding option is designed to provide reasonable performance and maximum compatibility with different versions of DB2, DB2 for MVS, DB2 for OS/390, DB2 UDB, and DB2 for OS/400. Optionally, administrators can use the FAST method when running the Create Package utility and creating packages in many target collections. The FAST option should not be used with DB2 for MVS and DB2 UDB for NT due to known incompatibilities. This VT_BSTR property defaults to NORM. This VT_BOOL type property defaults to VARIANT_FALSE, don't process binary fields as character fields.
DBPROP_DB2OLEDB_COMMITCTRL	This property indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. This VT_BOOL type property defaults to VARIANT_TRUE (auto commit).
DBPROP_DB2OLEDB_DEFAULTSCH	The name of the Collection where the OLE DB Provider for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The OLE DB Provider uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection. For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a VT_BSTR value for Default Schema, then the OLE DB Provider uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DBPROP_DB2OLEDB_HOSTCCSID	The character code set identifier (CCSID) matching the data as represented on the host. This property is required when processing binary data as character data. Unless the DBPROP_DB2OLEDB_BINASCHAR property ID is set to true, character data is converted based on the host column CCSID and default ANSI code page. This VT_I4 type property defaults to 37 (U.S./Canada).

DBPROP_DB2OLEDB_HOSTGCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This property is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. When accessing DB2 for OS/390 or DB2 for MVS, a value must be specified for GCCSID if the "MIXED DATA" field (7) of the DB2 installation panel for Application Programming Defaults (DSNTIPF) is set to "YES". This property is currently supported only by the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2 or later and by the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 3 or later. This property only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This VT_I4 type property defaults to indicating that mixed CCSID conversions are not supported.
DBPROP_DB2OLEDB_HOSTMCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This property is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. When accessing DB2 for OS/390 or DB2 for MVS, a value must be specified for MCCSID if the "MIXED DATA" field (7) of the DB2 installation panel for Application Programming Defaults (DSNTIPF) is set to "YES". This property is currently supported only by the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2 or later and by the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 3 or later. This property only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This VT_I4 type property defaults to 0 indicating that mixed CCSID conversions are not supported.
DBPROP_DB2OLEDB_ISOLATION	This property determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This VT_BSTR type property defaults to NC.
DBPROP_DB2OLEDB_LOCALLU	When LU 6.2 (SNA) is selected for the Network Transport Library, this property is the name of the local LU alias configured in the SNA server. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_NETADDRESS	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target host computer. This parameter indicates the IP address or TCP/IP host name alias associated with the DDM server on the host. The network address is required when connecting via TCP/IP.
DBPROP_DB2OLEDB_NETPORT	When TCP/IP has been selected for the Network Transport Library, this property is used to locate the target DDM service access port when connecting via TCP/IP. This parameter represents the TCP/IP port used for communication with the DDM service on the host. This VT_BSTR type property defaults to 446.

DBPROP_DB2OLEDB_NETTYPE	This property which represents the dynamic link library used for transport designates whether the provider connects via SNA LU 6.2 or TCP/IP for network communication. The possible values for this parameter are TCP/IP or SNA. If TCPIP is selected, then values for Network Address (DBPROP_DB2OLEDB_NETADDRESS) and Network Port (DBPROP_DB2OLEDB_NETPORT) are required. If SNA is selected, then values for APPC Local LU Alias (DBPROP_DB2OLEDB_LOCALLU, APPC Mode Name (DBPROP_DB2OLEDB_APPCMODE), and APPC Remote LU Alias (DBPROP_DB2OLEDB_REMOTELU) are required. This VT_BSTR type property defaults to SNA.
DBPROP_DB2OLEDB_PACKAGECOL	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be the same as the Default Schema (DBPROP_DB2OLEDB_DEFAULTSCH). The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The OLE DB Provider for DB2 will create packages dynamically in the location to which the user points using the this property ID. This is a VT_BSTR type property.
DBPROP_DB2OLEDB_PCCODEPAGE	The PC Code Page property ID indicates the code page to be used on the PC for character code conversion. This property is required when processing binary data as character data. Unless DBPROP_DB2OLEDB_BINASCHAR is set to true, character data is converted based on the default ANSI code page configured in Windows. If this parameter is set to Binary or 65535, then no character code conversions will take place. This VT_I4 type property defaults to 1252 (Latin 1).
DBPROP_DB2OLEDB_READONLY	A property indicates whether the OLE DB Provider creates a read-only data source. When this property is true, the user has only read access to objects such as tables and cannot perform certain operations (INSERT, UPDATE, and DELETE, for example). This VT_BOOL type property defaults to VARIANT_TRUE.
DBPROP_DB2OLEDB_REMOTELU	When LU 6.2 (SNA) is selected for the Network Transport Library (DBPROP_DB2OLEDB_NETTYPE), this property is the name of the remote LU alias configured in the SNA server. This VT_BSTR type property has no default value.
DBPROP_DB2OLEDB_TPNAME	This property represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. This VT_BSTR type property has no default value.

OLE DB Data Source Property Support in the OLE DB Provider for DB2

The following table summarizes the standard OLE DB version 2.0 data source properties from the DBPROP_DATASOURCE property set that are supported by the version of the Microsoft® OLE DB Provider for DB2 included with Host Integration Server 2000:

OLE DB Property ID	Comments
DBPROP_CURRENTCATALOG	The name of the current catalog. This property is the derived from the Initial Catalog parameter when configuring data sources. An application can use the CATALOGS schema rowset to enumerate catalogs. This VT_BSTR type property defaults to the value configured in the data source for Initial Catalog.

ADO Object, Method, Property, and Collection Support

ActiveX® Data Objects (ADO) version 2.0 defines a number of objects, methods, properties, and collections.

The Microsoft® OLE DB Provider for AS/400 and VSAM supports the ADO objects, methods, properties, and collections that are appropriate for an OLE DB data provider accessing a non-SQL host file system. The following topics provide detailed information on ADO support:

- ADO Object Support in the OLE DB Provider for AS/400 and VSAM
- ADO Method Support in the OLE DB Provider for AS/400 and VSAM
- ADO Property Support in the OLE DB Provider for AS/400 and VSAM
- ADO Collection Support in the OLE DB Provider for AS/400 and VSAM

The Microsoft® OLE DB Provider for DB2 supports the ADO objects, methods, properties, and collections that are appropriate for an OLE DB data provider accessing an SQL database. The following topics provide detailed information on ADO support:

- ADO Object Support in the OLE DB Provider for DB2
- ADO Method Support in the OLE DB Provider for DB2
- ADO Property Support in the OLE DB Provider for DB2
- ADO Collection Support in the OLE DB Provider for DB2

The Microsoft® ODBC Driver for DB2 supports the ADO objects, methods, properties, and collections that are appropriate for an ODBC driver accessing an SQL database. The following topics provide detailed information on ADO support:

- ADO Object Support in the ODBC Driver for DB2
- ADO Method Support in the ODBC Driver for DB2
- ADO Property Support in the ODBC Driver for DB2
- ADO Collection Support in the ODBC Driver for DB2

ADO Object Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the ADO version 2.0 objects that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM.

ADO object	Support
Collection	Yes, most methods
Command	Yes, some methods, some properties, and all collections
Connection	Yes, some methods, some properties, and all collections
Error	Yes, some properties
Field	Yes, all methods, properties, and collections
Parameter	No
Recordset	Yes, most methods, most properties, and all collections

The Parameter object will be supported by a later version of the OLE DB Provider for AS/400 and VSAM.

ADO Method Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the ADO version 2.0 object methods that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM.

ADO object	Method	Support
Collection	**Append**	No
	Clear	Yes
	Delete	Yes
	Item	Yes
	Refresh	Yes
Command	**CreateParameter**	No
	Cancel	No
	Execute	Yes, but options must be **adCmdText**
Connection	**BeginTrans**	No
	Cancel	No
	Close	Yes
	CommitTrans	No
	Execute	Yes, but options must be **adCmdText**
	Open	Yes
	OpenSchema	Yes
	RollbackTrans	No
Field	AppendChunk	Yes
	GetChunk	Yes
	ReadFromFile	No
	WriteToFile	No
Parameter	**AppendChunk**	No
Recordset	AddNew	Yes
	Cancel	No
	CancelBatch	Yes
	CancelUpdate	Yes
	Clone	Yes
	Close	Yes
	Delete	Yes
	Find	Yes
	GetRows	Yes
	Move	Yes
	MoveFirst	Yes
	MoveLast	Yes
	MoveNext	Yes
	MovePrevious	Yes
	NextRecordset	No
	Open	Yes
	Requery	Yes
	Resync	No
	Save	Yes
	Seek	No
	Supports	Yes
	Update	Yes
	UpdateBatch	Yes

Note that the **Collection** object is a special case, representing a collection of other ADO objects. These collection objects support several methods:

- **Append** to add an object to a collection
- **Clear** to empty all objects from a collection
- **Delete** to remove a single object from a collection
- **Item** to return a specific member object of a collection by name or ordinal number
- **Refresh** to update the objects in a collection to reflect objects available from and specific to the OLE DB provider

ADO Property Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the ADO version 2.0 object properties that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM.

ADO object	Property	Support
Command	ActiveConnection	Yes
	CommandText	Yes
	CommandTimeout	No
	CommandType	Yes
	Prepared	No
	State	Yes
Connection	Attributes	Yes
	CommandTimeout	No
	ConnectionString	Yes
	ConnectionTimeout	No
	CursorLocation	Yes.
	DefaultDatabase	No
	IsolationLevel	No
	Mode	Yes
	Provider	Yes
	State	Yes
	Version	Yes
Error	Description	Yes
	HelpContext	No
	HelpFile	No
	NativeError	Yes
	Number	Yes
	Source	Yes
	SQLState	No
Field	ActualSize	Yes
	Attributes	Yes
	DataFormat	No
	DefinedSize	Yes
	Name	Yes
	NumericScale	Yes
	OriginalValue	Yes
	Precision	Yes
	Type	Yes
	UnderlyingValue	Yes
	Value	Yes
Parameter	**Attributes**	No
	Direction	No
	Name	No
	NumericScale	No
	Precision	No
	Size	No
	Type	No
	Value	No
Recordset	**AbsolutePage**	No
	AbsolutePosition	No
	ActiveCommand	Yes

	ActiveConnection	Yes
	BOF	Yes
	Bookmark	Yes
	CacheSize	Yes
	CursorLocation	Yes
	CursorType	Yes
	DataMember	No
	DataSource	No
	EditMode	Yes
	EOF	Yes
	Filter	Yes
	Index	No
	Locktype	Yes
	MarshalOptions	No
	MaxRecords	No
	PageCount	No
	PageSize	No
	RecordCount	No
	Sort	Yes
	Source	Yes
	State	Yes
	Status	Yes
	StayInSync	No

ADO Collection Support in the OLE DB Provider for AS/400 and VSAM

The following table summarizes the ADO version 2.0 object collections that are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM.

ADO object	Collection	Support
Command	**Parameters**	No
	Properties	Yes
Connection	**Errors**	Yes
	Properties	Yes
Field	**Properties**	Yes
Parameter	**Properties**	No
Recordset	**Fields**	Yes
	Properties	Yes

ADO Command Object in the OLE DB Provider for AS/400 and VSAM

The ADO **Command** object is a definition of a specific command that is executed against an OLE DB data source.

Command objects are used to create a **Recordset** object and obtain records, execute a bulk operation, or manipulate the structure of a database. When using the Microsoft® OLE DB Provider for AS/400 and VSAM, some collections, methods, or properties of a **Command** object may generate an error when called.

The primary purpose of the **Command** object in the context of the OLE DB Provider for AS/400 and VSAM is to issue AS/400 Command Language (CL) commands for execution by the remote OS/400 DDM target server. For a listing of legal DDM command-line strings, see the *AS/400 DDM User's Guide* published by IBM.

The following **Command** object methods, properties, and collections are supported by the current version of the OLE DB Provider for AS/400 and VSAM:

Name	Comment
Execute method	Evaluates command text as a text string (only supported *Options* parameter for this method is **adCmdText**, which indicates that this is not an SQL command).
ActiveConnection property	Sets or returns the information used to establish a connection to a data source (see notes following).
CommandText property	Sets or returns the command text to be executed.
CommandType property	Sets or returns the type of command in a **CommandText** property.
State property	Describes the current state of an object.
Properties collection	Collections of properties on the command.

The **Execute** method executes a command and returns a **Recordset** object, if appropriate. You can use the **Command** object to open tables or execute DDM commands on a remote DDM server. If errors occur, they can be examined with the **Errors** collection on the **Connection** object.

A **Command** object can be created independently of a previously defined **Connection** object by setting the **ActiveConnection** property of the **Command** object to a valid connection string (see the **ConnectionString** property of the **Connection** object for the proper syntax). ADO still creates a **Connection** object, but it does not assign that object to an object variable. However, if multiple **Command** objects are to be associated with the same connection, the **Connection** object should be explicitly created and opened. This assigns the **Connection** object to an object variable. If the **ActiveConnection** property of the **Command** object is not set to this object variable, ADO creates a new **Connection** object for each **Command** object, even if the same connection string is used.

The **ActiveConnection** property associates an open connection with a **Command** object. The **CommandText** property defines the text version of a command. The syntax for the string in the **CommandText** property when used with the OLE DB Provider for AS/400 and VSAM is as follows:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

The **CommandType** property specifies the type of command described in the **CommandText** property prior to execution in order to optimize performance. The **CommandType** property must be set to **adCmdText** for use with OLE DB Provider for AS/400 and VSAM.

The **Command** object can also be used to open a data file after a **Connection** object has been opened and the **ActiveConnection** property has been set to this open connection. The **CommandText** property defines the data file to open (an **EXEC OPEN *DataSetName*** statement, for example, where *DataSetName* represents a valid data file or library member on the host). You must set the **CommandType** property to **adCmdText** for use with the OLE DB Provider for AS/400 and VSAM. If you open a host data file from a **Command** object, then the data file is opened as read-only. This results from the limitation that no

argument or option is passed by ADO that supplies a parameter describing whether the data set should be opened as read-only or updateable.

ADO Connection Object in the OLE DB Provider for AS/400 and VSAM

The ADO **Connection** object represents an open connection to an OLE DB data source. The **Provider** property sets the OLE DB provider. The connection can be configured before opening the data source by setting the **ConnectionString** properties. The **Version** property determines the version of the ADO implementation in use.

The **Open** method establishes the physical connection to the data source and the **Close** method terminates the connection. If errors occur, they can be examined with the **Errors** collection.

The following methods, properties, and collections for the **Connection** object are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

Name	Comment
Close method	Closes a connection to a data source.
Execute method	Evaluates command text as a table name (only supported *Options* parameter for this method is **adCmdTable**).
Open method	Opens a connection to a data source and may optionally pass **ConnectionString** parameters with this method. (the only supported *Options* parameter for this method is **adCmdText**).
OpenSchema method	Obtains database schema information from the OLE DB provider.
Attributes property	Indicates one or more characteristics supported for a given **Connection** object.
ConnectionString property	Contains the information used to establish a connection to a data source (see note following).
CursorLocation property	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
Mode property	Indicates the available permissions for modifying data in a connection.
Provider property	Sets or returns the name of the provider for a connection.
State property	Describes the current state of an object.
Version property	Returns the version number of the ADO implementation in use.
Errors collection	Collections of **Error** objects on the connection.
Properties collection	Collections of properties on the connection.

Note that the information needed to establish a connection to a data source can be set in the **ConnectionString** property or passed as part of the **Open** method. In either case, this information must be in a specific format for use with the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2. This information is either a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons.

ADO supports several standard ADO-defined arguments for the **ConnectionString** property as follows:

Argument	Description
Data Source	Name of the data source for the connection. This argument is optional when using the OLE DB Provider for AS/400 and VSAM.
File Name	Name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed and is not supported by the OLE DB Provider for AS/400 and VSAM.
Location	The Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	Valid mainframe or AS/400 password to use when opening the connection. This password is used by Host Integration Server 2000 to validate that the user can log on to the target host system and has appropriate access rights to the file.
Provider	Name of the provider to use for the connection. To use the OLE DB Provider for AS/400 and VSAM, the Provider string must be set to "SNAOLEDB." To use the OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB."
User ID	Valid mainframe or AS/400 user name to use when opening the connection. This user name is used by Host Integration Server 2000 to validate that the user can log on to the target host system and has appropriate access rights to the file.

The OLE DB Provider for AS/400 and VSAM also supports a number of provider-specific arguments, some of which will default to values in the registry. These arguments are as follows:

Argument	Description
BinAsChar	This parameter indicates whether to process binary fields as character fields (default is 0; don't process binary fields as character fields). This parameter is equivalent to the DBPROP_SNAOLEDB_BINASCHAR OLE DB property ID.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37). This parameter is equivalent to the DBPROP_SNAOLEDB_HOSTCCSID OLE DB property ID.
Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files. This parameter is equivalent to the DBPROP_SNAOLEDB_LIBRARY OLE DB property ID.
HCDFileName	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400. This parameter is equivalent to the DBPROP_SNAOLEDB_HCDPATH OLE DB property ID.
LocalLU	The name of the local LU alias configured in the SNA server. This parameter is equivalent to the DBPROP_SNAOLEDB_LOCALLU OLE DB property ID.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server 2000 configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes. This parameter is equivalent to the DBPROP_SNAOLEDB_APPCMODE OLE DB property ID.
NetAddr	When TCP/IP is selected for the Network Transport Library, this parameter indicates the IP address of the host. This parameter is equivalent to the DBPROP_SNAOLEDB_NETADDRESS OLE DB property ID.
NetPort	When TCP/IP is selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446. This parameter is equivalent to the DBPROP_SNAOLEDB_NETPORT OLE DB property ID.
NetLib	This parameter determines whether TCP/IP or SNA APPC are used for network communication. The possible values for this parameter are TCPIP or SNA; the default value is SNA. This parameter is equivalent to the DBPROP_SNAOLEDB_NETTYPE OLE DB property ID.
PCCodePage	The character code page to use on the computer. If this argument is omitted, the default value is set to Latin 1 (1252). This parameter is equivalent to the DBPROP_SNAOLEDB_PCCODEPAGE OLE DB property ID.
RDB	The Remote DataBase name for OS/400. You only need to specify this value if it is different from the remote LU alias configured in the SNA server.
Repair Host Keys	This parameter indicates whether the OLE DB provider should repair any host key values set in the registry; the default is false. This parameter is equivalent to the DBPROP_SNAOLEDB_REPAIRKEY OLE DB property ID.
RemoteLU	The name of the remote LU alias configured in the Host Integration Server 2000 computer. This parameter is equivalent to the DBPROP_SNAOLEDB_REMOTELU OLE DB property ID.
StrictVal	This parameter indicates whether strict validation should be used; the default is false. This parameter is equivalent to the DBPROP_SNAOLEDB_STRICTVAL OLE DB property ID.

A sample **ConnectionString** for use with the OLE DB Provider for AS/400 and VSAM follows:

```
Conn.Provider="SNAOLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PCCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

When opening a connection object in ADO 2.0, you must specify the Prompt connection property. For example, the following is valid with ADO 1.5 and ADO 2.0 and will prompt the user for **ConnectionString** properties.

```
Conn.ConnectionString = "Provider=SNAOLEDB
Conn.Properties("PROMPT")=adPromptAlways
Conn.Open
```

A sample **Open** method call with these parameters follows:

```
RS.Open "library/member",Conn,2,1,1
```

The last three parameters to the **Open** method correspond with the *CursorType* (the **adOpenDynamic** enum is 2, for example), *LockType* (the **adLockReadOnly** enum is 1, for example), and *Options* (**adCmdText** is 1, which indicates that the Source name should be evaluated as a table name). The *Options* parameter must be set to **adCmdText** (1) when used with a data source name with OLE DB Provider for AS/400 and VSAM.

The allowable values for CCSID when using SNANLS (SNA National Language Support) for character code conversions (the default) are as follows:

EBCDIC character set	CCSID value
Arabic	20420
Binary (No Conversion)	65535
Chinese (Simplified)	935
Chinese (Traditional)	937
Cyrillic (Russian)	20880
Cyrillic (Serbian, Bulgarian)	21025
Denmark/Norway (Euro)	1142
Denmark/Norway	20277
Finland/Sweden (Euro)	1143
Finland/Sweden	20278
France (Euro)	1147
France	20297
Germany (Euro)	1141
Germany	20273
Greek (Modern)	875
Greek	20423
Hebrew	20424
Icelandic (Euro)	1149
Icelandic	20871
International (Euro)	1148
International	500
Italy (Euro)	1144
Italy	20280
Japanese (English-lower)	931
Japanese (Extend English)	939
Japanese (Extend Katakana)	930
Japanese (Katakana)	290

Japanese (Katakana-Kanji)	5026
Japanese (Latin-Kanji)	5035
Korean	933
Latin America/Spain (Euro)	1145
Latin America/Spain	20284
Latin-1 Open System (Euro)	20924
Latin-1 Open System	1047
Multilingual/ROECE (Latin-2)	870
Thai	20838
Turkish (Latin-5)	1026
Turkish	20905
U.S./Canada (Euro)	1140
U.S./Canada	37
United Kingdom (Euro)	1146
United Kingdom	20285

Note that SNANLS conversion uses the locale configured for the data sources using data links. For more information on SNANLS, see the SDK documentation on SNA National Language Support.

ADO Error Object in the OLE DB Provider for AS/400 and VSAM

The ADO **Error** object contains details about data access errors pertaining to a single operation involving ADO. You can read the properties of an **Error** object to obtain specific details about each error.

The **Error** object does not support any methods or collections; however, the **Errors** collection supported by other objects provides the standard **Collection** methods (**Clear** and **Delete**). **Error** objects are automatically appended to the **Errors** collection by the OLE DB Provider when they occur. The following **Error** object properties are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

Property Name	Comment
Description	The text of the error alert that is returned based on the minor error code (specific to the OLE DB Provider for AS/400 and VSAM) contained in the **Error** object resulting from an error.
NativeError	A Long integer value of the error code returned by the OLE DB Provider for AS/400 and VSAM.
Number	The Long integer value of the error constant.
Source	A string that indicates the name of the object or application that originally generated an error.

ADO Field Object in the OLE DB Provider for AS/400 and VSAM

The ADO **Field** object represents a column of data with a common data type. Each **Field** object corresponds to a column in a **Recordset** object.

The following **Field** object methods, properties, and collections are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

Name	Comment
AppendChunk method	Appends data to a large text or binary data **Field** object.
GetChunk method	Returns all or portions of the contents of a large text or binary data **Field** object.
ActualSize property	Actual length of a field's value.
Attributes property	One or more characteristics supported for a given **Field** object.
DefinedSize property	Defined size of a **Field** object.
Name property	Name of the **Field** object.
NumericScale property	Scale of numeric values in a **Field** object for numeric data.
OriginalValue property	Value of a **Field** object that existed in the record before changes were made.
Precision property	Degree of precision for numeric values in a **Field** object for numeric data.
Type property	Operational type or data type for a **Field** object.
UnderlyingValue property	Current value of a **Field** object.
Value property	Value assigned to a **Field** object in a **Recordset**.
Properties collection	Collections of properties on the field.

ADO Recordset Object in the OLE DB Provider for AS/400 and VSAM

The ADO **Recordset** object represents the entire set of records from a base table. At any time, the **Recordset** object refers to only one record within the set as the current record.

The following **Recordset** object methods, properties, and collections are supported by the current version of the Microsoft® OLE DB Provider for AS/400 and VSAM:

Name	Comment
AddNew method	Creates a new record for an updateable **Recordset** object.
CancelBatch method	Cancels a pending batch update.
CancelUpdate method	Cancels any changes made to a current record or to a new record prior to calling the **UpdateBatch** method.
Clone method	Creates a duplicate **Recordset** object from an existing **Recordset** object.
Close method	Closes an open object and any dependent objects.
Delete method	Deletes the current record in an open **Recordset** object or an object from a collection.
Find method	Finds the next record to match a condition (ADO 1.5 and later).
GetRows method	Retrieves multiple records of a **Recordset** into an array.
Move method	Moves the position of the current record in a **Recordset** object.
MoveFirst method	Moves to the first record in a specified **Recordset**.
MoveLast method	Moves to the last record in a specified **Recordset**.
MoveNext method	Moves to the next record in a specified **Recordset**.
MovePrevious method	Moves to the previous record in a specified **Recordset**.
Open method	Opens a cursor on a **Recordset**.
Requery method	Updates the data in a **Recordset** object by re-executing the query on which the object is based (equivalent to calling the **Close** and **Open** methods in succession).
Save method	Saves a **Recordset** in a file or **Stream** object.
Seek method	
Supports method	Determines whether a specified **Recordset** object supports a particular type of function.
Update method	Saves any changes you make to the current record of a **Recordset** object.
UpdateBatch method	Writes all pending batch updates to disk.
ActiveCommand property	Returns the **Command** object that created the specified **Recordset**.
ActiveConnection property	Sets or returns the **Connection** object that the specified **Recordset** object currently belongs.
BOF property	Indicates whether the current record position is before the first record in a **Recordset** object.
Bookmark property	Returns a bookmark that uniquely identifies the current record in a **Recordset** object or sets the current record in a **Recordset** object identified by a valid bookmark.
CacheSize property	Sets or returns the number of records from a **Recordset** object that are cached locally in memory.
CursorLocation property	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
CursorType property	Sets or returns the type of cursor used in a **Recordset** object. Only the **adOpenDynamic CursorType** is supported by the current version of the OLE DB Provider for AS/400 and VSAM.
EditMode property	Indicates the editing status of the current record type.
EOF property	Indicates whether the current record position is after the last record in a **Recordset** object.
Filter property	Indicates a filter for data in a **Recordset** (revised in ADO 1.5 and later).

LockType property	Sets or returns the types of locks placed on records during editing. All four lock types (**adLockReadOnly**, **adLockOptimistic**, **adLockPessimistic**, and **adLockBatchOptimistic**) are supported by the OLE DB Provider for AS/400 and VSAM. Note that the OLE DB provider internally maps **adLockPessimistic** to a locktype of **adLockBatchOptimistic**.
Sort property	Indicates the column names and order to sort data in a **Recordset** object (new property in ADO 1.5 and later).
Source property	Sets or returns the source (table name or command object) for the data in a **Recordset**.
State property	Describes the current state of an object.
Status property	Indicates the status of the current record with respect to batch updates or other bulk operations.
Fields collection	Collections of fields on the **Recordset**.
Properties collection	Collections of properties on the **Recordset**.

The syntax supported by the OLE DB Provider for AS/400 and VSAM to open a recordset (table) using the **Recordset.Open** method is:

```
EXEC OPEN TableName
```

where *TableName* represents one of the following host file naming conventions.

Host file type	File naming convention
VSAM Data Sets	DATASETNAME.FILENAME
Partitioned Data Sets	DATASETNAME.FILENAME(MEMBER)
OS/400 Files	LIBRARY/FILE
OS/400 Files	LIBRARY/FILE.NAME
OS/400 File Members	LIBRARY/FILE(MEMBER)
OS/400 File Members	LIBRARY/FILE.NAME(MEMBER)

Note that if a member of a library contains a dot in the member name, the member name must be surrounded by double quotes. For example, if the member name is NAMES.DAT, the proper syntax for command text used for the **Recordset.Open** method is:

```
RecordSet.Open "EXEC OPEN LIBRARY/FILE(""NAMES.DAT"")",...
```

The full path to the mainframe data set must be specified. In the example above, there are two path elements (LIBRARY/FILE) and one name element (NAMES.DAT).

Whenever a data set is allocated, it is given a unique name composed of one or more segments. Each segment of the data set name is joined by periods and represents a level of qualification. For example, the following data set has four segments that comprise the fully-qualified data set name (three path elements and one name element):

```
SAMPLES.DEMO.KSDS.TITLES
```

The high-level qualifier is SAMPLES. The low-level qualifier is TITLES. Each segment can be from 1-8 characters in length (the first character must be alphabetic, while the remainder can be alphanumeric or hyphens). The full data set name must be no more than 44 characters in length and contain no more than 22 segments.

The Recordset **Bookmark** method is supported for all AS/400 physical and logical files, as well as the following mainframe file types:

- KSDS if the file has a unique key
- RRDS if the file has a unique key

The Recordset **AddNew** method can be used on ESDS files on the AS/400 only when you are positioned at the end of the **Recordset** object (file). With Alternate Index files on the AS/400, the **AddNew** method can be used to add records when at the end of the **Recordset** object or by key. With KSDS or RRDR files on the mainframe, the **AddNew** method adds new records by key.

To use the Recordset **Find** method or the **Filter** property, an AS/400 logical file, an AS/400 keyed physical file, a mainframe KSDS file with a unique key, or a mainframe RRDS file with a unique key must be used. If these methods or properties are used on an AS/400 nonkeyed physical file or any other mainframe file type, then the method fails.

The Recordset **Sort** property is used with an open **Recordset** object based on an AS/400 physical file. The **Sort** property enables the user to indicate which logical view to apply to an AS/400 physical file. The logical view must be a valid index specified in the description of the AS/400 physical file. The logical view is provided by the AS/400 logical file. The OLE DB Provider for AS/400 and VSAM responds to the **Sort** property request by first closing the open physical file, and then opening the logical file that points back to the data in the physical file.

The **Recordset Sort** property is only supported on AS/400 hosts. If the user opens a **Recordset** object based on an AS/400 logical file, then there is probably no need to use **Recordset**.**Sort**. For performance reasons, applications should be written to open the AS/400 logical file first, because the overhead is so much greater when opening a physical file first.

ADO Object Support in the OLE DB Provider for DB2

The following table summarizes the ADO version 2.0 objects that are supported by the current version of the Microsoft® OLE DB Provider for DB2.

ADO object	Support
Collection	Yes, most methods
Command	Yes, some methods, some properties, and all collections
Connection	Yes, some methods, some properties, and all collections
Error	Yes, some properties
Field	Yes, no methods, most properties, and all collections
Parameter	Yes, most methods, most properties, and all collections
Recordset	Yes, most methods, most properties, and all collections

The **Parameter** object will be supported by a later version of the OLE DB Provider for DB2.

ADO Method Support in the OLE DB Provider for DB2

The following table summarizes the ADO version 2.0 object methods that are supported by the current version of the Microsoft® OLE DB Provider for DB2.

ADO object	Method	Support
Collection	**Append**	No
	Clear	Yes
	Delete	Yes
	Item	Yes
	Refresh	Yes
Command	**CreateParameter**	Yes
	Cancel	No
	Execute	Yes
Connection	**BeginTrans**	Yes
	Cancel	No
	Close	Yes
	CommitTrans	Yes
	Execute	Yes
	Open	Yes
	OpenSchema	Yes
	RollbackTrans	Yes
Field	**AppendChunk**	No
	GetChunk	No
	ReadFromFile	No
	WriteToFile	No
Parameter	**AppendChunk**	No
Recordset	AddNew	Yes
	Cancel	No
	CancelBatch	Yes
	CancelUpdate	Yes
	Clone	Yes
	Close	Yes
	Delete	Yes
	Find	No
	GetRows	Yes
	Move	Yes
	MoveFirst	Yes
	MoveLast	No
	MoveNext	Yes
	MovePrevious	No
	NextRecordset	No
	Open	Yes
	Requery	Yes
	Resync	No
	Save	Yes
	Seek	No
	Supports	Yes
	Update	Yes
	UpdateBatch	Yes

Note that the **Collection** object is actually a special case, representing a collection of other ADO objects. These collection objects support several methods:

- **Append** to add an object to a collection

- **Clear** to empty all objects from a collection
- **Delete** to remove a single object from a collection
- **Item** to return a specific member object of a collection by name or ordinal number
- **Refresh** to update the objects in a collection to reflect objects available from and specific to the OLE DB provider

ADO Property Support in the OLE DB Provider for DB2

The following table summarizes the ADO version 2.0 object properties that are supported by the current version of the Microsoft® OLE DB Provider for DB2.

ADO object	Property	Support
Command	ActiveConnection	Yes
	CommandText	Yes
	CommandTimeout	No
	CommandType	Yes
	Prepared	Yes
	State	Yes
Connection	Attributes	Yes
	CommandTimeout	No
	ConnectionString	Yes
	ConnectionTimeout	No
	CursorLocation	Yes.
	DefaultDatabase	No
	IsolationLevel	Yes. Note that versions of the OLE DB Provider for DB2 supplied with SNA Server 4.0 did not support this property. **IsolationLevel** was specified in the Default Isolation parameter in the connection string or Data Links dialog. This connection string parameter is not supported by the OLE DB provider supplied with Host Integration Server.
	Mode	Yes
	Provider	Yes
	State	Yes
	Version	Yes
Error	Description	Yes
	HelpContext	No
	HelpFile	No
	NativeError	Yes
	Number	Yes
	Source	Yes
	SQLState	Yes
Field	ActualSize	Yes
	Attributes	Yes
	DataFormat	No
	DefinedSize	Yes
	Name	Yes
	NumericScale	Yes
	OriginalValue	Yes
	Precision	Yes
	Type	Yes
	UnderlyingValue	Yes
	Value	Yes
Parameter	**Attributes**	Yes
	Direction	Yes
	Name	Yes
	NumericScale	Yes
	Precision	Yes

	Size	Yes
	Type	Yes
	Value	Yes
Recordset	**AbsolutePage**	No
	AbsolutePosition	No
	ActiveCommand	Yes
	ActiveConnection	Yes
	BOF	Yes
	Bookmark	Yes
	CacheSize	Yes
	CursorLocation	Yes
	CursorType	Yes
	DataMember	No
	DataSource	No
	EditMode	Yes
	EOF	Yes
	Filter	No
	Index	No
	Locktype	Yes
	MarshalOptions	No
	MaxRecords	Yes
	PageCount	No
	PageSize	No
	RecordCount	No
	Sort	No
	Source	Yes
	State	Yes
	Status	Yes
	StayInSync	No

ADO Collection Support in the OLE DB Provider for DB2

The following table summarizes the ADO version 2.0 object collections that are supported by the current version of the Microsoft® OLE DB Provider for DB2.

ADO object	Collection	Support
Command	**Parameters**	Yes
	Properties	Yes
Connection	**Errors**	Yes
	Properties	Yes
Field	**Properties**	Yes
Parameter	**Properties**	Yes
Recordset	**Fields**	Yes
	Properties	Yes

ADO Command Object in the OLE DB Provider for DB2

The ADO **Command** object is a definition of a specific command that is to be executed against an OLE DB data source.

Command objects can be used to create a **Recordset** object and obtain records, to execute a bulk operation, or to manipulate the structure of a database. When using the Microsoft® OLE DB Provider for DB2, some collections, methods, or properties of a **Command** object may generate an error when called.

The primary purpose of the **Command** object in the context of the OLE DB Provider for DB2 is to issue SQL commands for execution by the remote DB2 target server. Legal SQL commands are documented for the target DB2 platforms in SQL Reference Guides published by IBM.

The following **Command** object methods, properties, and collections are supported by the current version of the OLE DB Provider for DB2:

Name	Comment
Execute method	Evaluates command text (only supported *Options* parameter for this method is **adCmdText**, which indicates that this is an SQL text command).
ActiveConnection property	Sets or returns the information used to establish a connection to a data source (see notes following).
CommandText property	Sets or returns the command text to be executed.
CommandType property	Sets or returns the type of command in a **CommandText** property.
State property	Describes the current state of an object.
Properties collection	Collections of properties on the command.

The **Execute** method executes a command and returns a **Recordset** object, if appropriate. The **Command** object can be used to open tables or execute SQL commands on a remote DB2 server. If errors occur, these can be examined with the **Errors** collection on the **Connection** object.

A **Command** object can be created independently of a previously defined **Connection** object by setting the **ActiveConnection** property of the **Command** object to a valid connection string (see the **ConnectionString** property of the **Connection** object for the proper syntax). ADO still creates a **Connection** object, but it does not assign that object to an object variable. However, if multiple **Command** objects are to be associated with the same connection, the **Connection** object should be explicitly created and opened. This assigns the **Connection** object to an object variable. If the **ActiveConnection** property of the **Command** object is not set to this object variable, ADO creates a new **Connection** object for each **Command** object, even if the same connection string is used.

The **ActiveConnection** property associates an open connection with a **Command** object. The **CommandText** property defines the text version of a command (**SELECT ALL FROM TABLE**, for example). The **CommandType** property specifies the type of command described in the **CommandText** property prior to execution in order to optimize performance. The **CommandType** property must be set to **adCmdText** for use with the OLE DB Provider for DB2.

ADO Connection Object in the OLE DB Provider for DB2

The ADO **Connection** object represents an open connection to an OLE DB data source. The **Provider** property sets the OLE DB provider to use. The connection can be configured before opening the data source by setting the **ConnectionString** properties. The version of the ADO implementation in use can be determined from the **Version** property.

The physical connection to the data source is established using the **Open** method and terminated with the **Close** method. If errors occur, these can be examined with the **Errors** collection.

The following **Connection** object methods, properties, and collections are supported by the current version of the Microsoft® OLE DB Provider for DB2:

Name	Comment
Close method	Closes a connection to a data source.
Execute method	Evaluates command text.
Open method	Opens a connection to a data source and may optionally pass **ConnectionString** parameters with this method.
OpenSchema method	Obtains database schema information from the OLE DB provider.
Attributes property	One or more characteristics supported for a given **Connection** object.
ConnectionString property	Contains the information used to establish a connection to a data source (see notes following).
CursorLocation property	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
IsolationLevel	Sets or returns the level of isolation for a Connection object. Note that versions of the OLE DB Provider for DB2 supplied with SNA Server 4.0 did not support this property.
Mode property	Indicates the available permissions for modifying data in a connection.
Provider property	Sets or returns the name of the provider for a connection.
State property	Describes the current state of an object.
Version property	Returns the version number of the ADO implementation in use.
Errors collection	Collections of **Error** objects on the connection.
Properties collection	Collections of properties on the connection.

The information needed to establish a connection to a data source can be set in the **ConnectionString** property or passed as part of the **Open** method. In either case, this information must be in a specific format for use with the OLE DB Provider for DB2. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons. ADO supports several standard ADO-defined arguments for the **ConnectionString** property as follows:

Argument	Description
Data Source	Name of the data source for the connection. This argument is optional when using the OLE DB Provider for DB2.
File Name	Name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed.
Location	The Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to main frame systems.
Password	Valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is equivalent to the DBPROP_AUTH_PASSWORD OLE DB property ID.
Provider	Name of the provider to use for the connection. To use the OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB."
User ID	Valid mainframe or AS/400 user name to use when opening the connection. This user name validates that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is equivalent to the DBPROP_AUTH_USERID OLE DB property ID.

The OLE DB Provider for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by OLE DB Provider for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier OLE DB Provider for DB2 included with SNA Server 4.0.

The arguments supported by the OLE DB Provider for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	Description
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDB NAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.

RemoteLU	The name of the remote LU alias configured in Host Integration Server.
TPName	The transaction program name when used with SQL/DS.
UOW	This parameter determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the OLE DB Provider for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
BindType	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Commit	This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter defaults to true (auto commit).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.

GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
IsoLvl	This parameter determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommited Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommited. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
LocalLU	The name of the local LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.

Mode Name	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
Net Addr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
Net Port	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
Net Lib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PC Code Page	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
Pkg Col	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
Read Only	When the Read Only parameter is set to true (ReadOnly=1), the OLE DB Provider for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
Remote LU	The name of the remote LU alias configured in Host Integration.
TP Name	The transaction program name when used with SQL/DS.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the OLE DB Provider for DB2 follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PcCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

When opening a connection object in ADO 2.0, you must specify the Prompt connection property. For example, the following is valid with ADO 1.5 and ADO 2.0 and will prompt the user for **ConnectionString** properties.

```
Conn.ConnectionString = "Provider=DB2OLEDB
Conn.Properties("PROMPT")=adPromptAlways
Conn.Open
```

A sample **Open** method call with these parameters follows:

```
RS.Open "Accounting",Conn,0,1,1
```

The last three parameters to the **Open** method correspond with the *CursorType* (the **adOpenForwardOnly** enum is 0, for example), *LockType* (the **adLockReadOnly** enum is 1, for example), and *Options* (**adCmdText** is 1, which indicates that the Source name should be evaluated as SQL text). The *Options* parameter must be set to **adCmdText** (1) when used with a data source name with OLE DB Provider for DB2.

The allowable values for CCSID when using SNANLS (SNA National Language Support) for character code conversions (the default) are as follows:

EBCDIC character set	**CCSID value**
Arabic	20420
Binary (No Conversion)	65535
Chinese (Simplified)	935
Chinese (Traditional)	937
Cyrillic (Russian)	20880
Cyrillic (Serbian, Bulgarian)	21025
Denmark/Norway (Euro)	1142
Denmark/Norway	20277
Finland/Sweden (Euro)	1143
Finland/Sweden	20278
France (Euro)	1147
France	20297
Germany (Euro)	1141
Germany	20273
Greek (Modern)	875
Greek	20423
Hebrew	20424
Icelandic (Euro)	1149
Icelandic	20871
International (Euro)	1148
International	500
Italy (Euro)	1144
Italy	20280
Japanese (English-lower)	931
Japanese (Extend English)	939
Japanese (Extend Katakana)	930
Japanese (Katakana)	290
Japanese (Katakana-Kanji)	5026
Japanese (Latin-Kanji)	5035
Korean	933
Latin America/Spain (Euro)	1145
Latin America/Spain	20284
Latin-1 Open System (Euro)	20924
Latin-1 Open System	1047
Multilingual/ROECE (Latin-2)	870
Thai	20838
Turkish (Latin-5)	1026

Turkish	20905
U.S./Canada (Euro)	1140
U.S./Canada	37
United Kingdom (Euro)	1146
United Kingdom	20285

Note that the SNANLS conversion uses the locale configured for the data sources using data links. For more information, see the SDK documentation on SNA National Language Support.

The allowable values for CCSID when using ANSI/OEM for character code conversions are:

ANSI/OEM character set	**CCSID value**
ANSI - Arabic	1256
ANSI - Baltic	1257
ANSI - Cyrillic	1251
ANSI - Eastern Europe	1250
ANSI - Greek	1253
ANSI - Hebrew	1255
ANSI - Latin I	1252
ANSI - Turkish	1254
ANSI/OEM - Korean (Extended Wansung)	949
ANSI/OEM - Japanese Shift-JIS	932
ANSI/OEM - Simplified Chinese GBK	936
ANSI/OEM - Traditional Chinese Big5	950
ANSI/OEM - Thai	874
ANSI/OEM - Vietnam	1258

ADO Error Object in the OLE DB Provider for DB2

The ADO **Error** object contains details about data access errors pertaining to a single operation involving ADO. You can read the properties of an **Error** object to obtain specific details about each error.

The **Error** object does not support any methods or collections; however, the **Errors** collection supported by other objects provides the standard **Collection** methods (**Clear** and **Delete**). **Error** objects are automatically appended to the **Errors** collection by the OLE DB Provider when they occur. The following **Error** object properties are supported by the current version of the Microsoft® OLE DB Provider for DB2:

Property Name	Comment
Description	The text of the error alert that is returned based on the minor error code (specific to the OLE DB Provider for DB2) co ntained in the **Error** object resulting from an error.
NativeError	A Long integer value of the error code returned by the OLE DB Provider for DB2.
Number	The Long integer value of the OLE DB error constant.
Source	A string that indicates the name of the object or application that originally generated an error.

ADO Field Object in the OLE DB Provider for DB2

The ADO **Field** object represents a column of data with a common data type. Each **Field** object corresponds to a column in a **Recordset** object.

The following **Field** object methods, properties, and collections are supported by the current version of the Microsoft® OLE DB Provider for DB2:

Name	Comment
ActualSize property	Actual length of a field's value.
Attributes property	One or more characteristics supported for a given **Field** object.
DefinedSize property	Defined size of a **Field** object.
Name property	Name of the **Field** object.
NumericScale property	Scale of numeric values in a **Field** object for numeric data.
OriginalValue property	Value of a **Field** object that existed in the record before changes were made.
Precision property	Degree of precision for numeric values in a **Field** object for numeric data.
Type property	Operational type or data type for a **Field** object.
UnderlyingValue property	Current value of a **Field** object.
Value property	Value assigned to a **Field** object in a **Recordset**.
Properties collection	Collections of properties on the field.

ADO Recordset Object in the OLE DB Provider for DB2

The ADO **Recordset** object represents the entire set of records from a base table. At any time, the **Recordset** object refers to only one record within the set as the current record.

The following **Recordset** object methods, properties, and collections are supported by the current version of the Microsoft® OLE DB Provider for DB2:

Name	Comment
AddNew method	Creates a new record for an updateable **Recordset** object.
CancelBatch method	Cancels a pending batch update.
CancelUpdate method	Cancels any changes made to a current record or to a new record prior to calling the **UpdateBatch** method.
Clone method	Creates a duplicate **Recordset** object from an existing **Recordset** object.
Close method	Closes an open object and any dependent objects.
Delete method	Deletes the current record in an open **Recordset** object or an object from a collection.
GetRows method	Retrieves multiple records of a **Recordset** into an array.
Move method	Moves the position of the current record in a **Recordset** object.
MoveFirst method	Moves to the first record in a specified **Recordset**.
MoveNext method	Moves to the next record in a specified **Recordset**.
Open method	Opens a cursor on a **Recordset**.
Requery method	Updates the data in a **Recordset** object by re-executing the query on which the object is based (equivalent to calling the **Close** and **Open** methods in succession).
Save method	Saves a **Recordset** in a file or **Stream** object.
Supports method	Determines whether a specified **Recordset** object supports a particular type of function.
Update method	Saves any changes you make to the current record of a **Recordset** object.
UpdateBatch method	Writes all pending batch updates to disk.
ActiveCommand property	Returns the **Command** object that created the specified **Recordset**.
ActiveConnection property	Sets or returns the **Connection** object that the specified **Recordset** object currently belongs.
BOF property	Indicates whether the current record position is before the first record in a **Recordset** object.
Bookmark property	Returns a bookmark that uniquely identifies the current record in a **Recordset** object or sets the current record in a **Recordset** object identified by a valid bookmark.
CacheSize property	Sets or returns the number of records from a **Recordset** object that are cached locally in memory.
CursorLocation property	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
CursorType property	Sets or returns the type of cursor used in a **Recordset** object. Only the **adOpenForwardOnly CursorType** is supported by the current version of the OLE DB Provider for DB2.
EditMode property	Indicates the editing status of the current record type.
EOF property	Indicates whether the current record position is after the last record in a **Recordset** object.
LockType property	Sets or returns the types of locks placed on records during editing. The OLE DB Provider for DB2 supports locks of type **adLockReadOnly** and **adLockPessimistic**.
MaxRecords property	Sets or returns the maximum number of records to return to a **Recordset** from a query.
Source property	Sets or returns the source (table name or command object) for the data in a **Recordset**.
State property	Describes the current state of an object.
Status property	Indicates the status of the current record with respect to batch updates or other bulk operations.
Fields collection	Collections of fields on the **Recordset**.
Properties collection	Collections of properties on the **Recordset**.

ADO Object Support in the ODBC Driver for DB2

The following table summarizes the ADO version 2.0 objects that are supported by the current version of the Microsoft® ODBC Driver for DB2.

ADO object	Support
Collection	Yes, most methods
Command	Yes, some methods, some properties, and all collections
Connection	Yes, some methods, some properties, and all collections
Error	Yes, some properties
Field	Yes, no methods, most properties, and all collections
Parameter	Yes, most methods, most properties, and all collections
Recordset	Yes, most methods, most properties, and all collections

The **Parameter** object will be supported by a later version of the ODBC Driver for DB2.

ADO Method Support in the ODBC Driver for DB2

The following table summarizes the ADO version 2.0 object methods that are supported by the current version of the Microsoft ODBC Driver for DB2.

ADO object	Method	Support
Collection	**Append**	No
	Clear	Yes
	Delete	Yes
	Item	Yes
	Refresh	Yes
Command	**CreateParameter**	Yes
	Cancel	No
	Execute	Yes
Connection	**BeginTrans**	Yes
	Cancel	No
	Close	Yes
	CommitTrans	Yes
	Execute	Yes
	Open	Yes
	OpenSchema	Yes
	RollbackTrans	Yes
Field	**AppendChunk**	No
	GetChunk	No
	ReadFromFile	No
	WriteToFile	No
Parameter	**AppendChunk**	No
Recordset	AddNew	Yes
	Cancel	No
	CancelBatch	Yes
	CancelUpdate	Yes
	Clone	Yes
	Close	Yes
	Delete	Yes
	Find	No
	GetRows	Yes
	Move	Yes
	MoveFirst	Yes
	MoveLast	Yes, when using a client-side cursor only.
	MoveNext	Yes
	MovePrevious	Yes, when using a client-side cursor only.
	NextRecordset	No
	Open	Yes
	Requery	Yes
	Resync	No
	Save	Yes
	Seek	No
	Supports	Yes
	Update	Yes
	UpdateBatch	Yes

Note that the **Collection** object is actually a special case, representing a collection of other ADO objects. These collection objects support several methods:

- **Append** to add an object to a collection

- **Clear** to empty all objects from a collection
- **Delete** to remove a single object from a collection
- **Item** to return a specific member object of a collection by name or ordinal number
- **Refresh** to update the objects in a collection to reflect objects available from and specific to the ODBC Driver

ADO Property Support in the ODBC Driver for DB2

The following table summarizes the ADO version 2.0 object properties that are supported by the current version of the Microsoft ODBC Driver for DB2.

ADO object	Property	Support
Command	ActiveConnection	Yes
	CommandText	Yes
	CommandTimeout	No
	CommandType	Yes
	Prepared	Yes
	State	Yes
Connection	Attributes	Yes
	CommandTimeout	No
	ConnectionString	Yes
	ConnectionTimeout	No
	CursorLocation	Yes.
	DefaultDatabase	No
	IsolationLevel	Yes. Note that versions of the ODBC Driver for DB2 supplied with SNA Server 4.0 did not support this property. **IsolationLevel** was specified in the DIL parameter in the connection string or Data Links dialog. This connection string parameter is not supported by the OLE DB provider supplied with Host Integration Server.
	Mode	Yes
	Provider	Yes
	State	Yes
	Version	Yes
Error	Description	Yes
	HelpContext	No
	HelpFile	No
	NativeError	Yes
	Number	Yes
	Source	Yes
	SQLState	Yes
Field	ActualSize	Yes
	Attributes	Yes
	DataFormat	No
	DefinedSize	Yes
	Name	Yes
	NumericScale	Yes
	OriginalValue	Yes
	Precision	Yes
	Type	Yes
	UnderlyingValue	Yes
	Value	Yes
Parameter	**Attributes**	Yes
	Direction	Yes
	Name	Yes
	NumericScale	Yes
	Precision	Yes

	Size	Yes
	Type	Yes
	Value	Yes
Recordset	**AbsolutePage**	No
	AbsolutePosition	No
	ActiveCommand	Yes
	ActiveConnection	Yes
	BOF	Yes
	Bookmark	Yes
	CacheSize	Yes
	CursorLocation	Yes
	CursorType	Yes
	DataMember	No
	DataSource	No
	EditMode	Yes
	EOF	Yes
	Filter	No
	Index	No
	Locktype	Yes
	MarshalOptions	No
	MaxRecords	Yes
	PageCount	No
	PageSize	No
	RecordCount	No
	Sort	No
	Source	Yes
	State	Yes
	Status	Yes
	StayInSync	No

ADO Collection Support in the ODBC Driver for DB2

The following table summarizes the ADO version 2.0 object collections that are supported by the current version of the Microsoft ODBC Driver for DB2.

ADO object	Collection	Support
Command	**Parameters**	Yes
	Properties	Yes
Connection	**Errors**	Yes
	Properties	Yes
Field	**Properties**	Yes
Parameter	**Properties**	Yes
Recordset	**Fields**	Yes
	Properties	Yes

ADO Command Object in the ODBC Driver for DB2

The ADO **Command** object is a definition of a specific command that is to be executed against an ODBC Driver data source.

Command objects can be used to create a **Recordset** object and obtain records, execute a bulk operation, or manipulate the structure of a database. When using the Microsoft® ODBC Driver for DB2, some collections, methods, or properties of a **Command** object can generate an error when called.

The primary purpose of the **Command** object in the context of the ODBC Driver for DB2 is to issue SQL commands for execution by the remote DB2 target server. Legal SQL commands are documented for the target DB2 platforms in SQL Reference Guides published by IBM.

The following **Command** object methods, properties, and collections are supported by the current version of the ODBC Driver for DB2:

Name	Comment
Execute method	Evaluates command text (only supported *Options* parameter for this method is **adCmdText**, which indicates that this is an SQL text command).
ActiveConnection property	Sets or returns the information used to establish a connection to a data source (see notes following).
CommandText property	Sets or returns the command text to be executed.
CommandType property	Sets or returns the type of command in a **CommandText** property.
State property	Describes the current state of an object.
Properties collection	Collections of properties on the command.

The **Execute** method executes a command and returns a **Recordset** object, if appropriate. The **Command** object can be used to open tables or execute SQL commands on a remote DB2 server. If errors occur, you can examine them with the **Errors** collection on the **Connection** object.

You can create a **Command** object independently of a previously defined **Connection** object by setting the **ActiveConnection** property of the **Command** object to a valid connection string (for the proper syntax, see the **ConnectionString** property of the **Connection** object). ADO still creates a **Connection** object, but it does not assign that object to an object variable. However, if multiple **Command** objects are to be associated with the same connection, the **Connection** object should be explicitly created and opened. This assigns the **Connection** object to an object variable. If the **ActiveConnection** property of the **Command** object is not set to this object variable, ADO creates a new **Connection** object for each **Command** object, even if the same connection string is used.

The **ActiveConnection** property associates an open connection with a **Command** object. The **CommandText** property defines the text version of a command (**SELECT ALL FROM TABLE**, for example). The **CommandType** property specifies the type of command described in the **CommandText** property prior to execution to optimize performance. The **CommandType** property must be set to **adCmdText** for use with the ODBC Driver for DB2.

ADO Connection Object in the ODBC Driver for DB2

The ADO **Connection** object represents an open connection to an ODBC data source. The **Provider** property sets the ODBC Driver to use. Setting the **ConnectionString** properties configures the connection before opening the data source. The **Version** property determines the version of the ADO implementation in use.

The **Open** method establishes the physical connection to the data source and the **Close** method terminates the connection. If errors occur, these can be examined with the **Errors** collection.

The following **Connection** object methods, properties, and collections are supported by the current version of the Microsoft ODBC Driver for DB2:

Name	Comment
Close method	Closes a connection to a data source.
Execute method	Evaluates command text.
Open method	Opens a connection to a data source and can optionally pass **ConnectionString** parameters with this method.
OpenSchema method	Obtains database schema information from the ODBC Driver.
Attributes property	One or more characteristics supported for a given **Connection** object.
ConnectionString property	Contains the information used to establish a connection to a data source (see notes following).
CursorLocation property	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
IsolationLevel	Sets or returns the level of isolation for a Connection object. Note that versions of the ODBC Driver for DB2 supplied with SNA Server 4.0 did not support this property.
Mode property	Indicates the available permissions for modifying data in a connection.
Provider property	Sets or returns the name of the provider for a connection.
State property	Describes the current state of an object.
Version property	Returns the version number of the ADO implementation in use.
Errors collection	Collections of **Error** objects on the connection.
Properties collection	Collections of properties on the connection.

The information needed to establish a connection to a data source can be set in the **ConnectionString** property or passed as part of the **Open** method. In either case, this information must be in a specific format for use with the ODBC Driver for DB2. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons. ADO supports several standard ADO-defined arguments for the **ConnectionString** property:

Argument	Description
Data Source	A required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file that is stored in the Program Files\Common Files\ODBC\Data Sources directory.
File Name	Name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed.
Location	The Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	Valid mainframe or AS/400 password for use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database.
Provider	Name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
User ID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database.

The ODBC Driver for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by ODBC Driver for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier ODBC Driver for DB2 included with SNA Server 4.0.

The arguments supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	**Description**
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.

NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, the alternative TP Name is set to 0X07F9F9F9.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.
UOW	Determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the ODBC Driver for DB2 supplied with SNA Server 4.0 are as follows:

Argument	Description
ACM	The Auto Commit Mode parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true (auto commit). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
BT	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.

DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
DIL	This Default Isolation Level parameter determines the isolation level provided for this data source in cases of simultaneous access to DB2 objects by multiple applications. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Please note that the ALL isolation level is not allowed. Users should set the isolation level to RS since this has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.

MCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.

RO	When the Read Only parameter is set to true (RO=1), the ODBC Driver for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the ODBC Driver for DB2 follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "UID=USERNAME;PWD=password",&_
  "LLU=LOCAL;RLU=DATABASE",&_
  "MN=QPCSUPP;CCSID=37;CP=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

When opening a connection object in ADO 2.0, you must specify the Prompt connection property. For example, the following syntax is valid with ADO 1.5 and ADO 2.0 and prompts the user for **ConnectionString** properties.

```
Conn.ConnectionString = "Provider=DB2OLEDB
Conn.Properties("PROMPT")=adPromptAlways
Conn.Open
```

A sample **Open** method call with these parameters follows:

```
RS.Open "Accounting",Conn,0,1,1
```

The last three parameters to the **Open** method correspond with the *CursorType* (the **adOpenForwardOnly** enum is 0, for example), *LockType* (the **adLockReadOnly** enum is 1, for example), and *Options* (**adCmdText** is 1, which indicates that the Source name should be evaluated as SQL text). The *Options* parameter must be set to **adCmdText** (1) when used with the a data source name with ODBC Driver for DB2.

The allowable values for CCSID when using SNANLS (SNA National Language Support) for character code conversions (the default) are as follows:

EBCDIC character set	CCSID value
Arabic	20420
Binary (No Conversion)	65535
Chinese (Simplified)	935
Chinese (Traditional)	937
Cyrillic (Russian)	20880
Cyrillic (Serbian, Bulgarian)	21025
Denmark/Norway (Euro)	1142
Denmark/Norway	20277
Finland/Sweden (Euro)	1143
Finland/Sweden	20278

France (Euro)	1147
France	20297
Germany (Euro)	1141
Germany	20273
Greek (Modern)	875
Greek	20423
Hebrew	20424
Icelandic (Euro)	1149
Icelandic	20871
International (Euro)	1148
International	500
Italy (Euro)	1144
Italy	20280
Japanese (English-lower)	931
Japanese (Extend English)	939
Japanese (Extend Katakana)	930
Japanese (Katakana)	290
Japanese (Katakana-Kanji)	5026
Japanese (Latin-Kanji)	5035
Korean	933
Latin America/Spain (Euro)	1145
Latin America/Spain	20284
Latin-1 Open System (Euro)	20924
Latin-1 Open System	1047
Multilingual/ROECE (Latin-2)	870
Thai	20838
Turkish (Latin-5)	1026
Turkish	20905
U.S./Canada (Euro)	1140
U.S./Canada	37
United Kingdom (Euro)	1146
United Kingdom	20285

Note that the SNANLS conversions use the locale configured for the data sources using data links. For more information, see the SDK documentation on SNA National Language Support.

The allowable values for CCSID when using ANSI/OEM for character code conversions are as follows:

ANSI/OEM character set	**CCSID value**
ANSI - Arabic	1256
ANSI - Baltic	1257
ANSI - Cyrillic	1251
ANSI - Eastern Europe	1250
ANSI - Greek	1253
ANSI - Hebrew	1255
ANSI - Latin I	1252
ANSI - Turkish	1254
ANSI/OEM - Korean (Extended Wansung)	949
ANSI/OEM - Japanese Shift-JIS	932
ANSI/OEM - Simplified Chinese GBK	936
ANSI/OEM - Traditional Chinese Big5	950
ANSI/OEM - Thai	874
ANSI/OEM - Vietnam	1258

ADO Error Object in the ODBC Driver for DB2

The ADO **Error** object contains details about data access errors pertaining to a single operation involving ADO. You can read the properties of an **Error** object to obtain specific details about each error.

The **Error** object does not support any methods or collections; however, the **Errors** collection supported by other objects provides the standard **Collection** methods (**Clear** and **Delete**). **Error** objects are automatically appended to the **Errors** collection by the ODBC Driver when they occur. The following **Error** object properties are supported by the current version of the Microsoft® ODBC Driver for DB2:

Property Name	Comment
Description	The text of the error alert that is returned based on the minor error code (specific to the ODBC Driver for DB2) contained in the **Error** object resulting from an error.
NativeError	A Long integer value of the error code returned by the ODBC Driver for DB2.
Number	The Long integer value of the ODBC error constant.
Source	A string that indicates the name of the object or application that originally generated an error.

ADO Field Object in the ODBC Driver for DB2

The ADO **Field** object represents a column of data with a common data type. Each **Field** object corresponds to a column in a **Recordset** object.

The following **Field** object methods, properties, and collections are supported by the current version of the Microsoft® ODBC Driver for DB2:

Name	Comment
ActualSize property	Actual length of a field's value.
Attributes property	One or more characteristics supported for a given **Field** object.
DefinedSize property	Defined size of a **Field** object.
Name property	Name of the **Field** object.
NumericScale property	Scale of numeric values in a **Field** object for numeric data.
OriginalValue property	Value of a **Field** object that existed in the record before changes were made.
Precision property	Degree of precision for numeric values in a **Field** object for numeric data.
Type property	Operational type or data type for a **Field** object.
UnderlyingValue	Current value of a **Field** object.
Value property	Value assigned to a **Field** object in a **Recordset**.
Properties collection	Collections of properties on the field.

ADO Recordset Object in the ODBC Driver for DB2

The ADO **Recordset** object represents the entire set of records from a base table. At any time, the **Recordset** object refers to only one record within the set as the current record.

The following **Recordset** object methods, properties, and collections are supported by the current version of the Microsoft® ODBC Driver for DB2:

Name	Comment
AddNew method	Creates a new record for an updateable **Recordset** object.
CancelBatch method	Cancels a pending batch update.
CancelUpdate method	Cancels any changes made to a current record or to a new record prior to calling the **UpdateBatch** method.
Clone method	Creates a duplicate **Recordset** object from an existing **Recordset** object.
Close method	Closes an open object and any dependent objects.
Delete method	Deletes the current record in an open **Recordset** object or an object from a collection.
GetRows method	Retrieves multiple records of a **Recordset** into an array.
Move method	Moves the position of the current record in a **Recordset** object.
MoveFirst method	Moves to the first record in a specified **Recordset**.
MoveLast method	Moves to the last record in a specified **Recordset**. This method is only supported when using a client-side cursor.
MoveNext method	Moves to the next record in a specified **Recordset**.
MovePrevious method	Moves to the previous record in a specified **Recordset**. This method is only supported when using a client-side cursor.
Open method	Opens a cursor on a **Recordset**.
Requery method	Updates the data in a **Recordset** object by re-executing the query on which the object is based (equivalent to calling the **Close** and **Open** methods in succession).
Save method	Saves a **Recordset** in a file or **Stream** object.
Supports method	Determines whether a specified **Recordset** object supports a particular type of function.
Update method	Saves any changes you make to the current record of a **Recordset** object.
UpdateBatch method	Writes all pending batch updates to disk.
ActiveCommand property	Returns the **Command** object that created the specified **Recordset**.
ActiveConnection property	Sets or returns the **Connection** object that the specified **Recordset** object currently belongs.
BOF property	Indicates whether the current record position is before the first record in a **Recordset** object.
Bookmark property	Returns a bookmark that uniquely identifies the current record in a **Recordset** object or sets the current record in a **Recordset** object identified by a valid bookmark.
CacheSize property	Sets or returns the number of records from a **Recordset** object that are cached locally in memory.
CursorLocation	Sets or returns the location of the cursor (whether the cursor is on the client or the server side).
CursorType	Sets or returns the type of cursor used in a **Recordset** object. Only the **adOpenForwardOnly CursorType** is supported by the current version of the ODBC Driver for DB2.
EditMode property	Indicates the editing status of the current record type.
EOF property	Indicates whether the current record position is after the last record in a **Recordset** object.
LockType property	Sets or returns the types of locks placed on records during editing. The ODBC Driver for DB2 supports locks of type **adLockReadOnly** and **adLockPessimistic**.
MaxRecords	Sets or returns the maximum number of records to return to a **Recordset** from a query.
Source property	Sets or returns the source (table name or command object) for the data in a **Recordset**.
State property	Describes the current state of an object.
Status property	Indicates the status of the current record with respect to batch updates or other bulk operations.
Fields collection	Collections of fields on the **Recordset**.

Properties collection	Collections of properties on the **Recordset**.

ADO Reference

This section provides reference information on specific ActiveX® Data Objects (ADO) methods, properties, and collections supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2.

ActiveCommand Property

The **ActiveCommand** property on a **Recordset** object indicates which **Command** object created the associated **Recordset** object. This property returns a Variant that contains a **Command** object. The default is a **Null** object reference.

```
currentCommand = recordset.ActiveCommand
```

Remarks

The **ActiveCommand** property is property is read-only. If a **Command** object was not used to create the current **Recordset**, then a **Null** object reference is returned.

Use this property to find the associated **Command** object when you are given only the resulting **Recordset** object.

ActiveConnection Property

The **ActiveConnection** property on a Command or **Recordset** object indicates to which **Connection** object the specified **Command** or **Recordset** object currently belongs. This property sets or returns a String containing the definition for a connection or a **Connection** object. The default is a **Null** object reference.

```
command.ActiveConnection = connectionString
activeConnectionString = recordset.ActiveConnection
```

Remarks

The **ActiveConnection** property is used to determine the **Connection** object over which the specified **Command** object will execute or the specified **Recordset** will be opened.

For **Command** objects, the **ActiveConnection** property is read/write.

If you attempt to call the **Execute** method on a **Command** object before setting the **ActiveConnection** property to an open **Connection** object or valid connection string, an error occurs.

Under Microsoft® Visual Basic®, setting the **ActiveConnection** property to **Nothing** disassociates the **Command** object from the current **Connection** and causes the OLE DB Provider to release any associated resources on the data source. You can then associate the **Command** object with the same or another **Connection** object. Some providers allow you to change the **ActiveConnection** property setting from one **Connection** to another, without having to first set the property to **Nothing**.

Closing the **Connection** object with which a **Command** object is associated sets the **ActiveConnection** property to **Nothing**. Setting this property to a closed **Connection** object generates an error.

For open **Recordset** objects or for **Recordset** objects whose **Source** property is set to a valid **Command** object, the **ActiveConnection** property is read-only. Otherwise, it is read/write.

You can set the **ActiveConnection** property to a valid **Connection** object or to a valid connection string. In this case, the OLE DB Provider creates a new **Connection** object using this definition and opens the connection. Additionally, the provider may set this property to the new **Connection** object to give you a way to access the **Connection** object for extended error information or to execute other commands.

If the *ActiveConnection* parameter of the **Open** method is used to open a **Recordset** object, the **ActiveConnection** property will inherit the value of the argument.

If the **Source** property of the **Recordset** object is set to a valid **Command** object variable, the **ActiveConnection** property of the **Recordset** inherits the setting of the **Command** object's **ActiveConnection** property.

The information needed to establish a connection to a data source can be set in the **ActiveConnection** property of a **Recordset** object or passed as part of the **Open** method on a **Recordset** object in the *ActiveConnection* parameter. In either case, this information must be in a specific format for use with the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, or the Microsoft®.ODBC Driver for DB2. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons.

ADO supports several standard ADO-defined arguments for the **ActiveConnection** property as follows:

Argument	**Description**
Data Source	Specifies the name of the data source for the connection. This argument is optional when using the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
File Name	Specifies the name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed. This argument is not supported by the OLE DB Provider for AS/400 and VSAM.

Location	The Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target host system and has appropriate access rights to the file.
Provider	Specifies the name of the provider to use for the connection. To use the OLE DB Provider for AS/400 and VSAM, the Provider string must be set to "SNAOLEDB". To use the OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB". To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
User ID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target host system and has appropriate access rights to the file.

ActiveConnection Property Support Using the OLE DB Provider for AS/400 and VSAM

The Microsoft® OLE DB Provider for AS/400 and VSAM also supports a number of provider-specific arguments, some of which have default values as specified in the table below. These arguments are as follows:

Argument	Description
BinAsChar	This parameter indicates whether to process binary fields as character fields (default is 0, don't process binary fields as character fields).
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files.
HCDFileName	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
RDB	The Remote DataBase name for OS/400. You only need to specify this value if it is different from the remote LU alias configured in Host Integration Server.
Repair Host Keys	This parameter indicates whether the OLE DB provider should repair any host key values set in the registry and defaults to false.
RemoteLU	The name of the remote LU alias configured in Host Integration Server.
StrictVal	This parameter indicates whether strict validation should be used and defaults to false.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** for use with the OLE DB Provider for AS/400 and VSAM follows:

```
Conn.Provider="SNAOLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PCCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

The &_ character combination is used for continuing long lines in Visual Basic.

When opening a connection object in ADO 2.0, you must specify the Prompt connection property. For example, the following is valid with ADO 1.5 and ADO 2.0 and will prompt the user for **ConnectionString** properties:

```
Conn.ConnectionString = "Provider=SNAOLEDB
Conn.Properties("PROMPT")=adPromptAlways
Conn.Open
```

ActiveConnection Property Support Using the OLE DB Provider for DB2

The Microsoft® OLE DB Provider for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by OLE DB Provider for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier OLE DB Provider for DB2 included with SNA Server 4.0.

The arguments supported by the OLE DB Provider for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDB NAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.

PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
RemoteLU	The name of the remote LU alias configured in Host Integration Server.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPName is set to 0X07F9F9F9.
UOW	This parameter determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the OLE DB Provider for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
BindType	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.

CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Commit	This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter defaults to true (auto commit).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
IsoLvl	This parameter determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommited Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommited. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
LocalLU	The name of the local LU alias configured in Host Integration Server.

MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
ReadOnly	When the Read Only parameter is set to true (ReadOnly=1), the OLE DB Provider for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RemoteLU	The name of the remote LU alias configured in Host Integration.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the OLE DB Provider for DB2 follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PcCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

The &_ character combination is used for continuing long lines in Visual Basic.

ActiveConnection Property Support Using the ODBC Driver for DB2

The Microsoft® ODBC Driver for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by ODBC Driver for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier ODBC Driver for DB2 included with SNA Server 4.0.

The arguments supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	**Description**
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.

MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPN is set to 0X07F9F9F9.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

UOW	Determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the ODBC Driver for DB2 supplied with Host Integration Server 2000 is as follows:

```
Conn.Provider="MSDASQL"
Conn.ConnectionString = "UID=USERNAME;PWD=password",&_
  "LLU=LOCAL;RLU=DATABASE",&_
  "MN=QPCSUPP;CCSID=37;CP=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

The arguments supported by the ODBC Driver for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
ACM	The Auto Commit Mode parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true (auto commit). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
BT	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.

CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
DIL	This Default Isolation Level parameter determines the isolation level provided for this data source in cases of simultaneous access to DB2 objects by multiple applications. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Please note that the ALL isolation level is not allowed. Users should set the isolation level to RS since this has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.

GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.

RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RO	When the Read Only parameter is set to true (RO=1), the ODBC Driver for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the ODBC Driver for DB2 supplied with SNA Server 4.0 is as follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "UID=USERNAME;PWD=password",&_
  "LLU=LOCAL;RLU=DATABASE",&_
  "MN=QPCSUPP;CCSID=37;CP=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

ActualSize Property

The **ActualSize** property on a **Field** object indicates the actual length of a field's value. This property returns a returns a Long value.

```
size = field.ActualSize
```

Remarks

The **ActualSize** property is used to return the actual length of a **Field** object's value. For all fields, the **ActualSize** property is read-only. If ADO cannot determine the length of the **Field** object's value, the **ActualSize** property returns **adUnknown**.

The **ActualSize** and **DefinedSize** properties on a **Field** object can be different. For example, a **Field** object with a declared type of **adVarChar** (variable character data type) and a maximum length of 50 characters returns a **DefinedSize** property value of 50, but the **ActualSize** property value it returns is the length of the data stored in the field for the current record.

AddNew Method

The **AddNew** method on a **Recordset** object creates a new record for an updatable **Recordset** object.

```
recordset.AddNew Fields, Values
```

Parameters

Fields
This optional parameter specifies a single name or an array of names or ordinal positions of the fields in the new record.

Values
This optional parameter specifies a single value or an array of values for the fields in the new record. If *Fields* is an array, *Values* must also be an array with the same number of members; otherwise, an error occurs. The order of field names must match the order of field values in each array.

Remarks

The **AddNew** method is used to create and initialize a new record. The **Supports** method can be used with **adAddNew** to verify whether records can be added to the current **Recordset** object.

After the **AddNew** method is called, the new record becomes the current record and remains current after the **Update** method is called. If the **Recordset** object does not support bookmarks, you may not be able to access the new record after you move to another record. Depending on your cursor type, you may need to call the **Requery** method to make the new record accessible.

If **AddNew** is called while editing the current record or while adding a new record, ADO calls the **Update** method to save any changes and then creates the new record.

The behavior of the **AddNew** method depends on the updating mode of the **Recordset** object and whether or not the *Fields* and *Values* arguments are passed.

In immediate update mode, the OLE DB Provider writes changes to the underlying data source after the **Update** method is called. In immediate update mode, calling the **AddNew** method without arguments sets the **EditMode** property to **adEditAdd**. The OLE DB Provider caches any field value changes locally. Calling the **Update** method posts the new record to the database and resets the **EditMode** property to **adEditNone**. If the *Fields* and *Values* arguments are passed, ADO immediately posts the new record to the database (no **Update** call is necessary) and the **EditMode** property value does not change (**adEditNone**).

In batch update mode, the OLE DB Provider caches multiple changes and writes them to the underlying data source only when the **UpdateBatch** method is called. In batch update mode, calling the **AddNew** method without arguments sets the **EditMode** property to **adEditAdd**. The OLE DB Provider caches any field value changes locally. Calling the **Update** method adds the new record to the current **Recordset** object and resets the **EditMode** property to **adEditNone**, but the OLE DB Provider does not post the changes to the underlying database until the **UpdateBatch** method is called. If the *Fields* and *Values* arguments are passed, ADO sends the new record to the provider for storage in a cache and the **UpdateBatch** method must be called to post the new record to the underlying database.

AppendChunk Method

The **AppendChunk** method on a **Field** object appends data to a large text or binary data **Field** object.

```
field.AppendChunk Data
```

Parameters

Data
This parameter specifies a Variant containing the data to be appended to the **Field** object.

Remarks

The **AppendChunk** method is used on a **Field** object to fill it with long binary or character data. In situations where system memory is limited, the **AppendChunk** method can be used to manipulate long values in portions rather than in their entirety.

If the **adFldLong** bit in the **Attributes** property of a **Field** object is set to **True**, the **AppendChunk** method can be used for that field.

The first **AppendChunk** call on a **Field** object writes data to the field, overwriting any existing data. Subsequent **AppendChunk** calls add to existing data. If you are appending data to one field and then set or read the value of another field in the current record, ADO assumes that you are finished appending data to the first field. If the **AppendChunk** method is called on the first field again, ADO interprets the call as a new **AppendChunk** operation and overwrites the existing data. Accessing fields in other **Recordset** objects (that are not clones of the first **Recordset** object) will not disrupt **AppendChunk** operations.

If there is no current record when the **AppendChunk** method is called on a **Field** object, an error occurs.

Attributes Property

The **Attributes** property on a **Field** object or a **Property** object in a **Properties** collection indicates one or more characteristics of an object. This property returns a Long value.

```
attribute = field.Attributes
```

Remarks

The **Attributes** property is used to return characteristics of **Field** objects or **Property** objects.

For a **Field** object, the **Attributes** property is read-only and its value can be the sum of any one or more of the **FieldAttributeEnum** values. The allowable **FieldAttributeEnum** values can be one of the following constants:

Enumeration	Value	Description
adFldMayDefer	0x2	This value indicates that the field is deferred, that is, the field values are not retrieved from the data source with the whole record, but only when you explicitly access them.
adFldUpdatable	0x4	This value indicates that you can write to the field.
adFldUnknownUpdatable	0x8	This value indicates that the provider cannot determine if you can write to the field.
adFldFixed	0x10	This value indicates that the field contains fixed-length data.
adFldIsNullable	0x20	This value indicates that the field accepts Null values.
adFldMayBeNull	0x40	This value indicates that you can read Null values from the field.
adFldLong	0x80	This value indicates that the field is a long binary field. This value also indicates that the **AppendChunk** and **GetChunk** methods on the **Field** object can be used.
adFldRowID	0x100	This value indicates that the field contains some kind of record identifier (record number, unique identifier, and so on).
adFldRowVersion	0x200	This value indicates that the field contains some kind of time or date stamp (often used to track updates).
adFldCacheDeferred	0x1000	This value indicates that the provider caches field values and that subsequent reads are done from the cache.

For a **Property** object, the **Attributes** property is read-only and its value can be the sum of any one or more of the **PropertyAttributesEnum** values. The allowable **PropertyAttributesEnum** values can be one of the following constants:

Enumeration	Value	Description
adPropNotSupported	0	This value indicates that the property is not supported by the provider.
adPropRequired	0x1	This value indicates that the user must specify a value for this property before the data source is initialized.
adPropOptional	0x2	This value indicates that the user does not need to specify a value for this property before the data source is initialized.
adPropRead	0x200	This value indicates that the user can read the property.
adPropWrite	0x400	This value indicates that the user can set the property.

BOF Property

The **BOF** property on a **Recordset** object indicates that the current record position is before the first record in a **Recordset** object. This property returns a Boolean value.

```
IsBOF = recordset.BOF
```

Remarks

The **BOF** property is used to determine whether a **Recordset** object contains records or whether you have gone beyond the limits of a **Recordset** object when you move from record to record.

The **BOF** property returns **True** if the current record position is before the first record and **False** if the current record position is on or after the first record.

If the **BOF** property is **True**, there is no current record.

If a **Recordset** object is opened containing no records, both the **BOF** and **EOF** properties are set to **True**, and the **Recordset** object's **RecordCount** property setting is zero. When a **Recordset** object is opened that contains at least one record, the first record is the current record and the **BOF** and **EOF** properties are **False**.

If the last remaining record in the **Recordset** object is deleted, the **BOF** and **EOF** properties may remain **False** until you attempt to reposition the current record.

This table below indicates which **Move** methods are allowed with different combinations of the **BOF** and **EOF** properties.

	MoveFirst MoveLast	**MovePrevious Move < 0**	**Move 0**	**MoveNext Move > 0**
BOF=True EOF=False	Allowed	Error	Error	Allowed
BOF=False EOF=True	Allowed	Allowed	Error	Error
Both True	Error	Error	Error	Error
Both False	Allowed	Allowed	Allowed	Allowed

> **Note** Executing a **Move** 0 method when the **BOF** property is **True** does not currently generate an error using the OLE DB Provider for AS/400 and VSAM.

> **Note** Allowing a **Move** method does not guarantee that the method will successfully locate a record; it only means that calling the specified **Move** method will not generate an error.

The following table shows what happens to the **BOF** and **EOF** property settings when various **Move** methods are called but are unable to successfully locate a record.

	BOF property	**EOF property**
MoveFirst MoveLast	Set to **True**	Set to **True**
Move 0	No change	No change
MovePrevious Move < 0	Set to **True**	No change
MoveNext Move > 0	No change	Set to **True**

Bookmark Property

The **Bookmark** property on a **Recordset** object returns a bookmark that uniquely identifies the current record in a **Recordset** object or sets the current record in a **Recordset** object to the record identified by a valid bookmark. This property sets or returns a Variant expression that evaluates to a valid bookmark.

```
FirstBookmark = recordset.Bookmark
recordset.Bookmark = PreviousBookmark
```

Remarks

The **Bookmark** property is used to save the position of the current record and return to that record at any time. Bookmarks are available only in **Recordset** objects (host tables) that support the bookmark feature.

When a **Recordset** object is opened, each of its records has a unique bookmark. To save the bookmark for the current record, assign the value of the **Bookmark** property to a variable. To quickly return to that record at any time after moving to a different record, set the **Recordset** object's **Bookmark** property to the value of that variable.

The user may not be able to view the value of the bookmark. Also, users should not expect bookmarks to be directly comparable —two bookmarks that refer to the same record may have different values.

If the **Clone** method is used to create a copy of a **Recordset** object, the **Bookmark** property settings for the original and the duplicate **Recordset** objects are identical and you can use them interchangeably. However, you cannot use bookmarks from different **Recordset** objects interchangeably, even if they were created from the same source or command.

Using the OLE DB Provider for AS/400 and VSAM, only some data sources can be bookmarked. Calling the **Supports** method with the **adBookmark** argument will indicate if the data source (table) can be bookmarked.

CacheSize Property

The **CacheSize** property on a **Recordset** object indicates the number of records from a **Recordset** object that are cached locally in memory. This property sets or returns a Long value that must be greater than zero. The default value for the **CacheSize** property is 1.

```
previousSize = recordset.CacheSize
recordset.CacheSize = 1
```

Remarks

The **CacheSize** property is used to control how many records the provider keeps in its buffer and how many to retrieve at one time into local memory. For example, if the **CacheSize** is 10, after first opening the **Recordset** object, the provider retrieves the first 10 records into local memory. As you move through the **Recordset** object, the provider returns the data from the local memory buffer. As soon as you move past the last record in the cache, the provider retrieves the next 10 records from the data source into the cache.

The value of the **CacheSize** property can be adjusted during the life of the **Recordset** object, but changing this value only affects the number of records in the cache after subsequent retrievals from the data source. Changing the property value alone will not change the current contents of the cache.

If there are fewer records to retrieve than the **CacheSize** property specifies, the provider returns the remaining records; no error occurs.

A **CacheSize** setting of zero is not allowed and returns an error. Non-bookmarkable files cannot have the **CacheSize** property set to greater than one, or an error will occur.

It is strongly recommended that a **CacheSize** of 1 be used with the OLE DB Provider for AS/400 and VSAM. If the **CacheSize** is set greater than 1, it is possible for the local data cached in memory to be out of date from changes made by other users on the host.

CancelBatch Method

The **CancelBatch** method on a **Recordset** object cancels a pending batch update.

```
recordset.CancelBatch AffectedRecords
```

Parameters

AffectedRecords

This optional parameter specifies an **AffectEnum** value that determines how many records the **CancelBatch** method will affect.

The **AffectEnum** value can be one of the following constants:

Enumeration	Value	Description
adAffectCurrent	1	This value cancels pending updates only for the current record.
adAffectGroup	2	This value cancels pending updates for records that satisfy the current Filter property setting. You must set the **Filter** property to one of the valid predefined constants to use this option.
adAffectAll	3	This value cancels pending updates for all the records in the **Recordset** object, including any hidden by the current **Filter** property setting. This value is the default.

Remarks

The **CancelBatch** method is used to cancel any pending updates in a **Recordset** object in batch update mode. If the **Recordset** object is in immediate update mode, calling **CancelBatch** without **adAffectCurrent** generates an error.

If you are editing the current record or are adding a new record when **CancelBatch** is called, ADO first calls the CancelUpdate method to cancel any cached changes, and then all pending changes in the recordset are canceled.

It is possible that the current record will be indeterminable after a **CancelBatch** call, especially if you were in the process of adding a new record. For this reason, it is prudent to set the current record position to a known location in the recordset after the **CancelBatch** method is called. For example, call the **MoveFirst** method.

If the attempt to cancel the pending updates fails because of a conflict with the underlying data (for example, a record has been deleted by another user), the provider returns warnings to the Errors collection but does not halt program execution. A run-time error occurs only if there are conflicts on all the requested records. The Filter property (**adFilterAffectedRecords**) and the **Status** property can be used to locate records with conflicts.

CancelUpdate Method

The **CancelUpdate** method on a **Recordset** object cancels any changes made to the current record or to a new record prior to calling the **Update** method.

```
recordset.CancelUpdate
```

Parameters

None.

Remarks

The **CancelUpdate** method is used to cancel any changes made to the current record or to discard a newly added record. You cannot undo changes to the current record or to a new record after the **Update** method is called unless the changes are part of a batch update that you can cancel with the CancelBatch method.

If you are adding a new record when the **CancelUpdate** method is called, the record that was current prior to the AddNew method call becomes the current record again.

If you have not changed the current record or added a new record, calling the **CancelUpdate** method generates an error.

Clear Method

The **Clear** method on a Collection object removes all of the objects in a collection.

```
collection.Clear
```

Parameters

None.

Remarks

The **Clear** method is used on the **Errors** collection to remove all existing **Error** objects from the collection. When an error occurs, ADO automatically clears the **Errors** collection and fills it with **Error** objects based on the new error. However, some properties and methods return warnings that appear as **Error** objects in the **Errors** collection but do not halt a program's execution. Before calling the **Resync**, **UpdateBatch**, or CancelBatch methods on a **Recordset** object or before setting the Filter property on a **Recordset** object, call the **Clear** method on the **Errors** collection. Doing so enables you to read the **Count** property of the **Errors** collection to test for returned warnings as a result of these specific calls.

Clone Method

The **Clone** method on a **Recordset** object creates a duplicate **Recordset** object from an existing **Recordset** object.

```
rstDuplicate = rstOriginal.Clone
```

Parameters

rstDuplicate
This object variable specifies the duplicate **Recordset** object to be created.
rstOriginal
This object variable specifies the **Recordset** object to be duplicated.

Remarks

The **Clone** method is used on a **Recordset** object to create multiple, duplicate **Recordset** objects, particularly if you want to be able to maintain more than one current record in a given set of records. Using the **Clone** method is more efficient than creating and opening a new **Recordset** object with the same definition as the original.

The current record of a newly created clone is set to the first record.

Changes made to one **Recordset** object are visible in all of its clones regardless of cursor type. However, after you execute **Requery** on the original **Recordset**, the clones will no longer be synchronized to the original.

Closing the original **Recordset** does not close its copies; closing a copy does not close the original or any of the other copies.

You can only clone a **Recordset** object that supports bookmarks. Bookmark values are interchangeable; that is, a bookmark reference from one **Recordset** object refers to the same record in any of its clones.

Close Method

The **Close** method on a **Connection** or **Recordset** object closes an open object and any dependent objects.

```
recordSet.Close
```

Parameters

None

Remarks

The **Close** method is used to close either a **Connection** object or a **Recordset** object to free any associated system resources. Closing an object does not remove it from memory; you may change its property settings and open it again later. To completely eliminate an object from memory, set the object variable to **Nothing**.

Using the **Close** method to close a **Connection** object also closes any active **Recordset** objects associated with the connection. A **Command** object associated with the **Connection** object you are closing will persist, but it will no longer be associated with a **Connection** object, that is, its **ActiveConnection** property will be set to **Nothing**.

You can later call the **Open** method to reestablish the connection to the same or another data source. While the **Connection** object is closed, calling any methods that require an open connection to the data source generates an error. Closing a **Connection** object while there are open **Recordset** objects on the connection rolls back any pending changes in all of the **Recordset** objects.

Using the **Close** method to close a **Recordset** object releases the associated data and any exclusive access you may have had to the data through this particular **Recordset** object. You can later call the **Open** method to reopen the recordset with the same or modified attributes. While the **Recordset** object is closed, calling any methods that require a live cursor generates an error.

If an edit is in progress while in immediate update mode, calling the **Close** method generates an error. The **Update** or CancelUpdate methods should be called first. If you close the **Recordset** object during batch updating, everything changes since the last **UpdateBatch** call is lost.

CommandText Property

The **CommandText** property on a **Command** object contains the text of a command that you want to issue against a provider. This property sets or returns a String value containing a provider command, such as an AS/400 Command Language (CL) command for execution by the remote OS/400 DDM target server or an SQL command for execution on a DB2 database server. The default value for the **CommandText** property is a zero-length string.

```
previousCommandtext = command.CommandText
command.CommandText = "EXEC COMMAND DDMCmd"
```

Remarks

The **CommandText** property is used to set or return the text of a **Command** object. When used with the OLE DB Provider for AS/400 and VSAM, the text can be an AS/400 CL command for execution by the remote OS/400 DDM target server or a request to open a table on a host (a remote DDM Server). When used with the OLE DB Provider for DB2, the text can be an SQL command for execution or a call to a stored procedure.

If the **Prepared** property of the **Command** object is set to **True** and the **Command** object is bound to an open connection when you set the **CommandText** property, ADO prepares the query when you call the **Execute** or **Open** methods.

Depending on the **CommandType** property setting, ADO may alter the **CommandText** property. The **CommandText** property can be read at any time to see the actual command text that ADO will use during execution.

The **CommandText** property defines the text version of a command. The syntax for the string in the **CommandText** property when used with the OLE DB Provider for AS/400 and VSAM is as follows:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

With the OLE DB Provider for AS/400 and VSAM, the **Command** object can also be used to open a data file after a **Connection** object has been opened and the **ActiveConnection** property has been set to this open connection. The **CommandText** property defines the data file to open. When used with the OLE DB Provider for AS/400 and VSAM, the syntax for the **CommandText** property string in this case is as follows:

```
EXEC OPEN DataSetName
```

where *DataSetName* represents a valid data file or library member on the host. If you open a host data file from a **Command** object, then the data file is opened as read-only. This results from the limitation that no argument or option is passed by ADO that supplies a parameter describing whether the data set should be opened as read-only or updatable.

The syntax for the string in the **CommandText** property when used with the OLE DB Provider for DB2 can be one of the following:

```
EXEC SQLStatement
```

where SQLStatement represents a valid SQL statement supported by DB2.

```
CALL StoredProcedure
```

where StoredProcedure represents a valid DB2 stored procedure on the database server.

The **CommandType** property specifies the type of command described in the **CommandText** property prior to execution in order to optimize performance. The **CommandType** property must be set to **adCmdText** for use with the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

CommandType Property

The **CommandType** property on a **Command** object Indicates the type of a **Command** object. This property sets or returns a **CommandTypeEnum** value.

```
oldType = command.CommandType
command.CommandType = newType
```

Remarks

The **CommandType** property is used to set or return the type of a **Command** object. This property specifies a **CommandTypeEnum** value that can be one of the following constants:

Enumeration	Value	Description
adCmdUnspecified	-1	This value indicates that the **CommandType** property has been unspecified.
adCmdText	1	This value evaluates the **CommandText** property as a textual definition of a command command or stored procedure call.
adCmdTable	2	This value evaluates the **CommandText** property as a table name. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adCmdStoredProc	4	This value evaluates the **CommandText** property as a stored procedure. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2. See remarks below regarding using stored procedures using the OLE DB Provider for DB2.
adCmdUnknown	8	This value indicates that the type of command in a **CommandText** property is not known. This is the default value.

The OLE DB Provider for AS/400 and VSAM and the OLE DB Provider for DB2 only support the **adCmdText** type for the **CommandType** property. If any other value for the **CommandType** property is set, errors will occur.

The OLE DB Provider for DB2 supports calling DB2 stored procedures. An application must use the CALL keyword before the SQL statement in order to execute a stored procedure. When using ADO, a **CommandType** property of **adCmdStoredProc** cannot be used for executing a stored procedure since ADO inserts an EXEC not CALL keyword before the command text. In order to execute a stored procedure using ADO, the **CommandType** property should be set to **adCmdText** and the CALL keyword should be used before the SQL statement containing the stored procedure to be executed.

ConnectionString Property

The **ConnectionString** property on a **Connection** object contains the information used to establish a connection to a data source. This property sets or returns a string value.

```
oldString = connection.ConnectionString
connection.ConnectionString = newString
```

Remarks

The **ConnectionString** property is used to specify a data source by passing a detailed connection string containing a series of *argument = value* statements separated by semicolons.

ADO supports several standard arguments for the **ConnectionString** property. Any other arguments are passed directly to the provider without any processing by ADO. This information must be in a specific format for use with the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, or the Microsoft® ODBC Driver for DB2. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons. ADO supports several standard ADO-defined arguments for the **ConnectionString** property as follows:

Argument	Description
DataSource	This argument specifies the name of the data source for the connection. This argument is optional when using the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
FileName	This argument specifies the name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed. This argument is not supported by the OLE DB Provider for AS/400 and VSAM.
Location	The Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	This argument specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target host system and has appropriate access rights to the file.
Provider	This argument specifies the name of the provider to use for the connection. To use the OLE DB Provider for AS/400 and VSAM, the Provider string must be set to "SNAOLEDB". To use the OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB". To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
Remote Provider	This argument specifies the name of a provider to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
Remote Server	This argument specifies the path name of a server to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
User ID	This argument specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target host system and has appropriate access rights to the file.

Note Not all of these parameters are required. The user can also be prompted for this information.

The OLE DB Provider for AS/400 and VSAM also supports a number of provider-specific arguments, some of which default have default values as specified in the table below. These arguments are as follows:

Argument	Description
BinAsChar	This parameter indicates whether to process binary fields as character fields (default is 0, don't process binary fields as character fields).
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).

Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files.
HCDFileName	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
RDB	The Remote DataBase name for OS/400. You only need to specify this value if it is different from the remote LU alias configured in Host Integration Server.
Repair Host Keys	This parameter indicates whether the OLE DB provider should repair any host key values set in the registry and defaults to false.
RemoteLU	The name of the remote LU alias configured in Host Integration Server.
StrictVal	This parameter indicates whether strict validation should be used and defaults to false.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** for use with the OLE DB Provider for AS/400 and VSAM follows:

```
Conn.Provider="SNAOLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PCCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

The &_ character combination is used for continuing long lines in Visual Basic.

When opening a connection object in ADO 2.0, you must specify the Prompt connection property. For example, the following is valid with ADO 1.5 and ADO 2.0 and will prompt the user for **ConnectionString** properties:

```
Conn.ConnectionString = "Provider=SNAOLEDB
Conn.Properties("PROMPT")=adPromptAlways
Conn.Open
```

The OLE DB Provider for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by OLE DB Provider for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier OLE DB Provider for DB2 included with SNA Server 4.0.

The arguments supported by the OLE DB Provider for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	Description
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.

RemoteLU	The name of the remote LU alias configured in Host Integration Server.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPName is set to 0X07F9F9F9.
UOW	This parameter determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the OLE DB Provider for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
BindType	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Commit	This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter defaults to true (auto commit).

DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
IsoLvl	This parameter determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommited Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommited. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
LocalLU	The name of the local LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.

ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
ReadOnly	When the Read Only parameter is set to true (ReadOnly=1), the OLE DB Provider for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RemoteLU	The name of the remote LU alias configured in Host Integration.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the OLE DB Provider for DB2 supplied with SNA Server 4.0 is as follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "User ID=USERNAME;Password=password",&_
  "LocalLU=LOCAL;RemoteLU=DATABASE",&_
  "ModeName=QPCSUPP;CCSID=37;PcCodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

The ODBC Driver for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by ODBC Driver for DB2 supplied with Host Integration Server 2000 differ from the

arguments supported by the earlier ODBC Driver for DB2 included with SNA Server 4.0.

The arguments supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	Description
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.

NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPN is set to 0X07F9F9F9.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.
UOW	Determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the ODBC Driver for DB2 supplied with Host Integration Server 2000 is as follows:

```
Conn.Provider="MSDASQL"
Conn.ConnectionString = "UID=USERNAME;PWD=password",&_
  "LLU=LOCAL;RLU=DATABASE",&_
  "MN=QPCSUPP;CCSID=37;CP=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

The arguments supported by the ODBC Driver for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
ACM	The Auto Commit Mode parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true (auto commit). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
BT	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.

DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
DIL	This Default Isolation Level parameter determines the isolation level provided for this data source in cases of simultaneous access to DB2 objects by multiple applications. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Please note that the ALL isolation level is not allowed. Users should set the isolation level to RS since this has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.

MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.

RO	When the Read Only parameter is set to true (RO=1), the ODBC Driver for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

Note Not all of these parameters are required. The user can also be prompted for this information.

A sample **ConnectionString** using the ODBC Driver for DB2 supplied with SNA Server 4.0 is as follows:

```
Conn.Provider="DB2OLEDB"
Conn.ConnectionString = "UID=USERNAME;PWD=password",&_
  "LLU=LOCAL;RLU=DATABASE",&_
  "MN=QPCSUPP;CCSID=37;CP=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

Note The &_ character combination is used for continuing long lines in Visual Basic.

After the **ConnectionString** property is set and the **Connection** object is opened, the provider may alter the contents of the property, for example, by mapping the ADO-defined argument names to their provider equivalents. Using the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2, three items are stripped from the **ConnectionString** after a **Connection** object is opened: Data Source, User ID, and Password.

The **ConnectionString** property automatically inherits the value used for the *ConnectionString* argument of the **Open** method on a **Connection** object, so you can override the current **ConnectionString** property during the **Open** method call. Therefore, the **ConnectionString** property of the **Connection** object can be set before opening the **Connection** object, or the *ConnectionString* parameter can be used to set or override the current connection parameters during the **Open** method call.

The **ConnectionString** property is read/write when the connection is closed and read-only when it is open.

If user and password information is set in both the **ConnectionString** property and in the optional *UserID* and *Password* parameters to the **Open** method, the results may be unpredictable. Such information should only be passed in either the **ConnectionString** property (or the *ConnectionString* parameter to the **Open** method call) or in the *UserID* and *Password* parameters.

There are a number of different ways to open a connection. The **Open** method can pass all of the appropriate connection information as part of the *ConnectionString* parameter or by setting the **ConnectionString** property of the **Connection** object, if this information is known in advance. The syntax in this case using the **ConnectionString** property is as follows:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.Provider="SNAOLEDB"
connection.ConnectionString = "User ID=USERNAME;Password=password;Local LU=LOCAL;Remote LU=DA
TABASE;ModeName=QPCSUPP;CCSID=37;CodePage=437"
Conn.Properties("PROMPT")=adPromptNever
Conn.Open
```

There is a lack of spaces after the semicolons in the string. If spaces are inserted after the semicolons, an error will occur.

The simplest form of a **ConnectionString** property that contains all necessary information is as follows:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.ConnectionString = "Provider=SNAOLEDB;Data Source=REMLU;User ID=USERNAME;Password=
password;ModeName=QPCSUPP"
```

Note The User ID and Password must be included. Note the lack of spaces after the semicolons in the string. If spaces are inserted after the semicolons, an error will occur.

In the case where you would like the user to input the connection information, the following syntax can be used. This syntax does not specify any connection information except the provider, which is always required unless this is set in the **ConnectionString** or **Provider** property of the **Connection** object:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.ConnectionString = "Provider=SNAOLEDB"
Conn.Properties("PROMPT")=adPromptAlways
connection.Open
```

This method of invoking the **Open** method automatically causes a dialog box to appear asking the user for the user name, password, and other necessary information.

CursorLocation Property

The **CursorLocation** property on a **Connection** object or **Recordset** object indicates the location of the cursor engine. This property sets or returns a Long value representing a **CursorLocationEnum**.

```
cursor = connection.CursorLocation
connection.CursorLocation = adUseServer
```

Remarks

The **CursorLocation** property is used to set or return the location of the cursor. This property can be set to one of the following **CursorLocationEnum** constants:

Enumeration	Value	Description
adUseNone	1	This value indicates no cursor location. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM.
adUseServer	2	This value indicates that the data provider or driver-supplied cursor is used.
adUseClient	3	This value indicates that a client-side cursor supplied by a local cursor library is to be used.
adUseClientBatch	3	For backward compatibility, this value indicates that a client-side cursor supplied by a local cursor library is to be used.

This property setting only affects connections established after the property has been set. Changing the **CursorLocation** property has no effect on existing connections.

This property is read/write on a **Connection** object or a closed **Recordset**, object and read-only on an open **Recordset**.

If the **CursorLocation** property is set to **adUseClient**, the recordset will be accessible as read-only, and recordset updates to the host are not possible. When the **CursorLocation** property is set to **adUseClient** (use the client cursor engine), the **Find** method, **Filter** property, and **Sort** property will work if MDAC 2.0 or higher is installed, but will not work properly with earlier versions of ADO.

CursorType Property

The **CursorType** property on a **Recordset** object indicates the type of the cursor engine. This property sets or returns a Long value representing a **CursorTypeEnum**.

```
oldType = recordset.CursorType
recordset.CursorType = newType
```

Remarks

The **CursorType** property is used to set or return the type of the cursor that the provider should use when opening the **Recordset**. This property can be one of the following enumerated values for **CursorTypeEnum**:

Enumeration	Value	Description
adOpenUnspecified	-1	This indicates an unspecified value for the *CursorType*. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
AdOpenForwardOnly	0	Specifying this value opens a forward-only-type cursor. This *CursorType* is identical to a static cursor, except that you can only scroll forward through records. This improves performance when only one pass through a **Recordset** is needed. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM.
AdOpenKeyset	1	Specifying this value opens a keyset-type cursor. This *CursorType is* similar to a dynamic cursor with a few exceptions. Records that other users delete are inaccessible from your **Recordset**. Data changes to existing records by other users are still visible, but records added by other users are not visible (cannot be seen). This value is not supported by the OLE DB Provider for AS/400 and VSAM.
AdOpenDynamic	2	Specifying this value opens a dynamic-type cursor. Additions, changes, and deletions by other users are visible, and all types of movement through the recordset are allowed, except for bookmarks if the provider does not support them. A dynamic cursor is the only *CursorType* supported by the OLE DB Provider for AS/400 and VSAM.
AdOpenStatic	3	Specifying this value opens a static-type cursor. A static cursor provides a static copy of a set of records that can be used to find data or generate reports. Additions, changes, or deletions by other users are not visible with a static cursor. This value is not supported by the OLE DB Provider for AS/400 and VSAM.

Note that the **Open** method on a **Recordset** object defaults this property to **adOpenForwardOnly**, a value that is mapped to **adOpenDynamic** by the OLE DB provider for AS/400 and VSAM.

This property setting only affects connections established after the property has been set. Changing the **CursorType** property has no effect on existing connections. The **CursorType** property is read/write when the **Recordset** is closed and read-only when it is open.

If a provider does not support the requested cursor type, the provider may return another cursor type. The **CursorType** property will change to match the actual cursor type in use when the **Recordset** object is open. To verify whether a specific returned cursor is supported, use the **Supports** method. When using the OLE DB Provider for AS/400 and VSAM, the **Supports** method returns **adMovePrevious** as true with **CursorType** set to **adOpenDynamic**. After you close the **Recordset**, the **CursorType** property reverts to its original setting.

DefinedSize Property

The **DefinedSize** property on a **Field** object indicates the defined size of a field object. This property returns a Long value that reflects the defined size of a field as a number of bytes.

```
size = field.DefinedSize
```

Remarks

The **DefinedSize** property is used to return the data capacity or length of a **Field** object. For all fields, the **DefinedSize** property is read-only. If ADO cannot determine the length of the **Field** object, the **ActualSize** property returns **adUnknown**.

The **ActualSize** and **DefinedSize** properties on a field object can be different. For example, a **Field** object with a declared type of **adVarChar** (variable character data type) and a maximum length of 50 characters returns a **DefinedSize** property value of 50, but the **ActualSize** property value it returns is the length of the data stored in the field for the current record.

Delete Method

The **Delete** method on a **Recordset** object deletes the current record or a group of records.

```
recordSet.Delete AffectedRecords
```

Parameters

AffectedRecords

This optional parameter specifies an **AffectEnum** value that determines how many records the **Delete** method will affect.

The **AffectEnum** value can be one of the following constants:

Enumeration	Value	Description
adAffectCurrent	1	This value deletes only the current record.
adAffectGroup	2	This value deletes the records that satisfy the current Filter property setting. You must set the **Filter** property to one of the valid predefined constants to use this option.

Remarks

The **Delete** method is used to mark the current record or a group of records in a **Recordset** object for deletion. If the **Recordset** object does not allow record deletion, an error occurs. If you are in immediate update mode, deletions occur in the database immediately. Otherwise, the records are marked for deletion from the cache and the actual deletion happens when the **UpdateBatch** method is called. (Use the **Filter** property to view the deleted records.)

Retrieving field values from the deleted record generates an error. After deleting the current record, the deleted record remains current until you move to a different record. After you move away from the deleted record, it is no longer accessible.

If you are in batch update mode, the CancelBatch method can be used to cancel a pending deletion or group of pending deletions.

If the attempt to delete records fails because of a conflict with the underlying data (for example, a record has already been deleted by another user), the data provider returns warnings to the **Errors** collection but does not halt program execution. A run-time error occurs only if there are conflicts on all the requested records.

Description Property

The **Description** property on a **Error** object describes the **Error** object. This property returns a String value that contains a description of the error.

```
errorString = currentError.Description
```

Remarks

The **Description** property on a **Error** object is used to obtain a short description of the error. Applications can display this property to alert the user to an error that the application does not want to handle. The string will come from either ADO or a data provider such as the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, or the Microsoft® ODBC Driver for DB2. The provider is responsible for passing specific error text to ADO.

ADO adds an **Error** object to the **Errors** collection for each provider error or warning it receives. An application can enumerate the **Errors** collection to trace the errors that the provider passes.

EditMode Property

The **EditMode** property on a **Recordset** object indicates the editing status of the current record. This property returns a Long value representing a **EditModeEnum**.

```
currentMode = recordset.EditMode
connection.CursorType = newType
```

Remarks

The **EditMode** property is used to return the editing status of the current recordset. This property can be one of the following enumerated values for **EditModeEnum**:

Enumeration	Value	Description
adEditNone	0	This value indicates that no editing operation is in progress.
adEditInProgress	1	This value indicates that data in the current record has been modified but not saved.
adEditAdd	2	This value indicates that the **AddNew** method has been called, and the current record in the copy buffer is a new record that has not been saved in the database.
adEditDelete	4	This value indicates that the current record has been deleted.

ADO maintains an editing buffer associated with the current record. The **EditMode** property indicates whether changes have been made to this buffer, or whether a new record has been created. Use the **EditMode** property to determine the editing status of the current record. You can test for pending changes if an editing process has been interrupted and determine whether you need to use the Update or CancelUpdate method.

See the AddNew method for a more detailed description of the **EditMode** property under different editing conditions.

EOF Property

The **EOF** property on a **Recordset** object indicates that the current record position is after the last record in a **Recordset** object. This property returns a Boolean value.

```
IsEOF = recordset.EOF
```

Remarks

The **EOF** property is used to determine whether a **Recordset** object contains records or whether you have gone beyond the limits of a **Recordset** object when you move from record to record.

The **EOF** property returns **True** if the current record position is after the last record and **False** if the current record position is on or before the last record.

If the **EOF** property is **True**, there is no current record.

If a **Recordset** object is opened containing no records, both the **BOF** and **EOF** properties are set to **True**, and the **Recordset** object's **RecordCount** property setting is zero. When a **Recordset** object is opened that contains at least one record, the first record is the current record and the **BOF** and **EOF** properties are **False**.

If the last remaining record in the **Recordset** object is deleted, the **BOF** and **EOF** properties may remain **False** until you attempt to reposition the current record.

This table below indicates which **Move** methods are allowed with different combinations of the **BOF** and **EOF** properties.

	MoveFirst MoveLast	**MovePrevious Move < 0**	**Move 0**	**MoveNext Move > 0**
BOF=True EOF=False	Allowed	Error	Error	Allowed
BOF=False EOF=True	Allowed	Allowed	Error	Error
Both True	Error	Error	Error	Error
Both False	Allowed	Allowed	Allowed	Allowed

Allowing a **Move** method does not guarantee that the method will successfully locate a record; it only means that calling the specified **Move** method will not generate an error.

The following table shows what happens to the **BOF** and **EOF** property settings when various **Move** methods are called, but are unable to successfully locate a record:

	BOF Property	**EOF Property**
MoveFirst MoveLast	Set to **True**	Set to **True**
Move 0	No change	No change
MovePrevious Move < 0	Set to **True**	No change
MoveNext Move > 0	No change	Set to **True**

Execute Method on Command Object

The **Execute** method on a **Command** object executes the statement specified in the **CommandText** property.

```
command.Execute( RecordsAffected, Parameters, Options )
set recordSet = command.Execute( RecordsAffected, Parameters,
Options )
```

Parameters

RecordsAffected
This optional parameter specifies a Long variable to which the provider returns the number of records that the operation affected.

Parameters
This optional parameter specifies a Variant array of parameter values passed with an SQL statement and is not used by the OLE DB Provider for AS/400 and VSAM.

Options
This optional parameter specifies a Long value representing a **CommandTypeEnum** value that indicates how the provider should evaluate the **CommandText** property of the **Command** object.

The **CommandTypeEnum** value can be one of the constants listed in the table following the Parameters section.

Constants used for CommandTypeEnum

Enumeration	Value	Description
adCmdUnspecified	-1	This value indicates that the **CommandText** property is unspecified. This value is not supported by the Microsoft ® OLE DB Provider for AS/400 and VSAM.
adCmdText	1	This value evaluates the **CommandText** property as a as a textual definition of a command or stored procedure call.
adCmdTable	2	This value evaluates the **CommandText** property as a table name. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
adCmdStoredProc	4	This value evaluates the **CommandText** property as a stored procedure name. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adCmdUnknown	8	This value indicates that the type of command in **CommandText** is not known. This value is the default. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

The default for *Option* is a**dCmdText** under the OLE DB Provider for AS/400 and VSAM and the OLE DB Provider for DB2.

Remarks

When used with the OLE DB Provider for AS/400 and VSAM, the **Execute** method on a **Command** object can be used to open tables or execute DDM commands on a remote DDM server. The *Options* parameter must be set to **adCmdText** for use with the OLE DB Provider for AS/400 and VSAM.

The primary purpose of the **Command** object in the context of the OLE DB Provider for AS/400 and VSAM is to issue AS/400 Command Language (CL) commands for execution by the remote OS/400 DDM target server. To invoke DDM commands on a remote DDM server, the **CommandText** property defines the text version of a command which must have been set to:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

Note that this method does not return the results or output of a remote DDM CL command. If the results or output of a remote command are to be captured, the ***DDMCmd*** statement to be executed must include syntax to redirect the command output to a file on the AS/400 host and then explicitly open this output file after the command has completed.

The **Execute** method on a **Command** object can also be used to open a data file after a **Connection** object has been opened and the **ActiveConnection** property has been set to this open connection. The **CommandText** property defines the data file to open and must be set to:

```
EXEC OPEN DataSetName
```

where *DataSetName* represents a valid data file or library member on the host. When used in this way, the **Execute** method returns a **Recordset** object. If you open a host data file from a **Command** object, then the data file is opened as read-only. This results from the limitation that no argument or option is passed by ADO that supplies a parameter describing whether the data set should be opened as read-only or updatable.

When used with the OLE DB Provider for DB2, the **Execute** method on a **Command** object can be used to execute SQL statements or call a stored procedure. The **CommandText** property defines the SQL statements to execute and must be set to one of the following:

```
EXEC SQLStatement
```

where SQLStatement represents a valid SQL statement supported by DB2.

```
CALL StoredProcedure
```

where StoredProcedure represents a valid DB2 stored procedure on the database server.

If errors occur, these can be examined with the **Errors** collection on the **Command** object.

Execute Method on Connection Object

The **Execute** method on a **Connection** object executes the statement specified in the **CommandText** property.

```
connection.Execute CommandText, RecordsAffected, Options
set recordset = connection.Execute( CommandText, RecordsAffected,
Options )
```

Parameters

RecordsAffected
This optional parameter specifies a Long variable to which the provider returns the number of records that the operation affected.

Options
This optional parameter specifies a Long value representing a **CommandTypeEnum** value that indicates how the provider should evaluate the **CommandText** property of the Command object.

The CommandTypeEnum value can be one of the following constants:

Enumeration	Value	Description
adCmdUnspecified	-1	This value indicates that the **CommandText** property is unspecified. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
adCmdText	1	This value evaluates the **CommandText** property as a textual definition of a command or stored procedure call.
adCmdTable	2	This value evaluates the **CommandText** property as a table name. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adCmdStoredProc	4	This value evaluates the **CommandText** property as a stored procedure name. This value is not supported by the OLE DB Provider for AS/400 and VSAM.
adCmdUnknown	8	This value indicates that the type of command in **CommandText** is not known. This value is the default. This value is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

Remarks

The **Execute** method on a **Connection** object can be used to open tables or execute DDM commands on a remote DDM server. The *Options* parameter must be set to **adCmdText** for use with the OLE DB Provider for AS/400 and VSAM.

The **Execute** method on a **Connection** object is primarily used to open a data file after a **Connection** object has been created. The **CommandText** property defines the data file to open and must be set to:

```
EXEC OPEN DataSetName
```

where *DataSetName* represents a valid data file or library member on the host. When used in this way, the **Execute** method returns a **Recordset** object. If you open a host data file from a **Connection** object, then the data file is opened as read-only. This results from the limitation that no argument or option is passed by ADO that supplies a parameter describing whether the data set should be opened as read-only or updatable. If a **Recordset** object is desired that is not read-only, then first create a **Recordset** object with the desired property settings, and use the **Recordset** object's **Open** method to return the open **Recordset**.

The **Execute** method on a **Connection** object can also be used to issue AS/400 Command Language (CL) commands for execution by the remote OS/400 DDM target server. To invoke DDM commands on a remote DDM server, the **CommandText** property defines the text version of a command which must have been set to:

```
EXEC COMMAND DDMCmd
```

where *DDMCmd* represents a valid OS/400 control language (CL) command. Note that only OS/400 CL commands are supported. These commands allow you to request functions from the OS/400 operating system. Some examples are the DLTF (Delete File) or DSPFFD (Display File Description) commands. These are the same commands that could be issued on the command line if you were connected to an AS/400 via a 5250 terminal session. See the 'OS/400 CL Reference for your platform for a detailed list of possible commands.

Note that this method does not return the results or output of a remote DDM CL command. If the results or output of a remote command are to be captured, the ***DDMCmd*** statement to be executed must include syntax to redirect the command output to a file on the AS/400 host and then explicitly open this output file after the command has completed.

If errors occur, these can be examined with the **Errors** collection on the **Connection** object.

Filter Property

The **Filter** property on a **Recordset** object indicates a filter for data in a **Recordset**. This property sets or returns a Variant value.

```
recordset.Filter = Criteria
```

Parameters

Criteria

This parameter specifies a Variant.

It can be either a criteria (a where clause) or one of the following enumerated values for **FilterGroupEnum**:

Enumeration	Value	Description
adFilterNone	0	No filter. This value removes the current filter and restores all records to view.
AdFilterPendingRecords	1	Use the pending records. This value allows viewing only those records that have changed but have not yet been sent to the server. This value is only applicable for batch update mode.
AdFilterAffectedRecords	2	Use only records affected by the last **Delete**, **Resync**, **UpdateBatch**, or **CancelBatch** call.
AdFilterFetchedRecords	3	Use the last fetched records. This value allows viewing the records in the current cache returned as a result of the last call to retrieve records (implying a resynchronization).
AdFilterConflictingRecords	5	Use the conflicting records. This value allows viewing only those records that failed the last batch update.

Remarks

The **Filter** property is not supported by the Microsoft® OLE DB Provider for DB2 or the Microsoft® ODBC Driver for DB2.

The **Filter** property is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM on certain files. In order to use the Recordset **Filter** property, an AS/400 logical file, an AS/400 keyed physical file, a mainframe KSDS file with a unique key, or a mainframe RRDS file with a unique key must be used. If this property is used on an AS/400 nonkeyed physical file or any other mainframe file type, then the method fails.

When the **Filter** property is used with a criteria, the where clause is a combination of triplets. Each triplet consists of a column name, an operator, and a literal value. These where clause triplets can be combined with **AND**s and **OR**s for more complex logical filters.

If *Criteria* is a single-condition where clause, then any operator can be used. The construction of a single-condition where clause consists of a column name (the database field), an operator (greater than or equal, for example), and a literal value.

Examples of a single-condition where clause is as follows:

- recordset.Filter = "LastName = 'Jones' "
- recordset.Filter ="Salary > 30000.0"

The *Criteria* argument can be a two-condition with the following restrictions:

- If the column name (the database field) is the same in both clauses, then the separate where clauses must define a contiguous range.
- If the column names are different, then the operators must be the same. If the operators are "LIKE", then the filtered region may be unexpected.

Examples of acceptable two-condition where clauses are as follows:

- recordset.Filter = "LastName = 'Jones' **AND** FirstName = 'Tom' "
- recordset.Filter = "Cost = 5000.00 **OR** Cost > 5000.00 "

The *Criteria* argument can be three or more conditions with the following restrictions:

- The operators must be the same for all conditions. If the operators are "LIKE", then the filtered region may be unexpected.

In all cases, if the "=" operator is used, then the column names specified in the where clause must be keyed columns in the file.

One restriction on these combinations is that **OR** clauses can only be used at the highest (major) level of the logical operation.

Examples of acceptable *Criteria* meeting these conditions are:

- recordset.Filter = "(Title='Manager' **AND** Salary>30000) **OR** (Title='Administrator' **AND** Salary>50000)"
- recordset.Filter ="Salary > 30000.0"

An example of illegal *Criteria* is:

- recordset.Filter = "(Title='Associate' **OR** Salary >30000) **AND** (Title='Administrator')"

The operator can also use wildcards (* or %) in character expressions as follows:

- recordset.Filter = "Lastname LIKE '*SMITH*'"

To determine if any records were found meeting the *Criteria*, the application should check the **Recordset EOF** property. If **EOF** is true, then no records were found meeting the where clause specified in the *Criteria* parameter.

If the CursorLocation property is set to **adUseClient** (use the client cursor engine), the **Filter** property will work if MDAC 2.0 or later is installed but will not work properly with earlier versions of ADO.

When operating on large VSAM files and only querying data on a subset of the records, using the **Filter** property is not desirable because of the performance impact. The entire VSAM file is transferred to the client for filtering. A better solution is to use the server cursor engine and the **Find** method.

Find Method

The **Find** method on a **Recordset** object locates or seeks to the next record in a **Recordset** object that meets a particular condition and makes this the current record. This method can be used to seek to a specific record in a **Recordset** object based on a where clause (similar to an SQL where clause) defined by the user.

```
recordset.Find Criteria, SkipRecords, SearchDirection, Start
```

Parameters

ADO supports four arguments for the **Find** method, but the last three arguments are optional and have default values as noted below:

Criteria
This BSTR parameter specifies the criteria used for locating or seeking to a record in a **Recordset** object.

SkipRecords
This optional parameter specifies a Long expression that indicates the number of records to skip (whether to skip the current record) when locating a record in a **Recordset** object. The default value for this argument is 0 (don't skip the current record). The first time a **Find** method is used, this argument is usually set to 0 (the default). On subsequent calls to this method to seek other records that meet the specified condition, this argument would normally be set to 1, to skip one record forward before finding the next record that matches the search *Criteria*. A negative value for this parameter is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM.

SearchDirection
This optional parameter is an enumeration that specifies the direction for the search.

It can be one of the following enumerated values for **SearchDirectionEnum**:

Enumeration	Value	Description
adSearchForward	0	Search forward from the current record.
AdSearchBackward	1	Search backward from the current record. This option is not supported by the OLE DB Provider for AS/400 and VSAM.

This optional argument defaults to **adSearchForward**.

Start
This optional parameter specifies the starting location for a search, which can be a bookmark or an enumeration indicating the current, first, or last record in a **Recordset** object. This argument is a Variant.

It can be either a bookmark or one of the following enumerated values for **BookmarkEnum**:

Enumeration	Value	Description
adBookmarkCurrent	0	The current record.
AdBookmarkFirst	1	The first record.
AdBookmarkLast	2	The last record.

This optional argument defaults to **adBookmarkCurrent**.

Remarks

The **Find** method is not supported by the Microsoft® OLE DB Provider for DB2 or the Microsoft® ODBC Driver for DB2.

The **Find** method is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM on certain files. In order to use the Recordset **Find** method, an AS/400 logical file, an AS/400 keyed physical file, a mainframe KSDS file with a unique key, or a mainframe RRDS file with a unique key must be used. If this method is used on an AS/400 nonkeyed physical file or any other mainframe file type, then the method fails.

The first parameter is the only required argument for the **Find** method. All of the other arguments are optional and have default values. This first argument is a single-condition where clause. The construction of a single-condition where clause consists of a column name (the database field), an operator (greater than or equal, for example), and a literal value.

Examples of acceptable single-condition where clauses are as follows:

- recordset.Find, "Cost > 10000.00"

- recordset.Find, "Cost < 100.00"
- recordset.Find, "Cost = 5000.00"
- recordset.Find, "LastName = 'Jones' "

Note that variables cannot be used as substitutes for literal values. If the file has multiple keys in the index, using the "=" operator will always fail since the values of all keys cannot be specified.

If the CursorLocation property is set to **adUseClient** (use the client cursor engine), the **Filter** method will work if MDAC 2.0 or later is installed, but will not work properly with earlier versions of ADO.

When operating on large VSAM files and only querying data on a subset of the records, using the **Filter** property is not desirable because of the performance impact. The entire VSAM file is transferred to the client for filtering. A better solution is to use the server cursor engine and the **Find** method.

GetChunk Method

The **GetChunk** method on a **Field** object returns all or a portion of the contents of a large text or binary data **Field** object.

```
variable = field.GetChunk( Size )
```

Parameters

Size
This parameter specifies a Long expression equal to the number of bytes or characters to be retrieved.

Return Values

A Variant.

Remarks

The **GetChunk** method on a **Field** object is used to retrieve part or all of its long binary or character data. In situations where system memory is limited, the **GetChunk** method can be used to manipulate long values in portions rather than in their entirety.

The data that a **GetChunk** method returns is assigned to a variable. If the *Size* parameter is greater than the remaining data, the **GetChunk** method returns only the remaining data without padding the variable with empty spaces. If the **Field** object is empty, the **GetChunk** method returns Null.

Each subsequent **GetChunk** method call retrieves data starting from where the previous **GetChunk** call left off. However, if you are retrieving data from one field and then set or read the value of another field in the current record, ADO assumes you are finished retrieving data from the first field. If the **GetChunk** method is called on the first field again, ADO interprets the call as a new **GetChunk** operation and starts reading from the beginning of the data. Accessing **Field** objects in other **Recordset** objects (that are not clones of the first **Recordset** object) will not disrupt **GetChunk** operations.

If the **adFldLong** bit in the **Attributes** property of a **Field** object is set to **True**, the **GetChunk** method can be used for that **Field** object.

If there is no current record when the **GetChunk** method is invoked on a **Field** object, error 3021 (no current record) occurs.

GetRows Method

The **GetRows** method on a **Recordset** object retrieves multiple records of a recordset into an array.

```
array = recordset.GetRows( Rows, Start, Fields )
```

Parameters

Rows

This optional parameter specifies a Long expression indicating the number of records to retrieve. The default value if this parameter is not specified is an **GetRowSetEnum** which is **adGetRowsRest** (value = -1).

Start

This optional parameter specifies the starting location for the record from which the **GetRows** operation should begin, which can be a bookmark or an enumeration indicating the current, first, or last record in a **Recordset** object. This argument is a Variant.

It can be either a bookmark or one of the following enumerated values for **BookmarkEnum**:

Enumeration	Value	Description
adBookmarkCurrent	0	The current record
adBookmarkFirst	1	The first record
adBookmarkLast	2	The last record

This optional argument defaults to **adBookmarkCurrent**.

Fields

This optional parameter is a Variant and specifies a single field name or ordinal position or an array of field names or ordinal position numbers. ADO returns only the data in these fields.

Return Values

A two-dimensional array.

Remarks

The **GetRows** method is used to copy records from a recordset into a two-dimensional array. The first subscript of the array identifies the field and the second array subscript identifies the record number. The array variable is automatically dimensioned to the correct size when the **GetRows** method returns the data.

If a value is not specified for the *Rows* parameter, the **GetRows** method automatically retrieves all the records in the **Recordset** object. If more records are requested than are available, **GetRows** returns only the number of available records.

If the **Recordset** object supports bookmarks, you can specify at which record the **GetRows** method should begin retrieving data by passing the value of that record's **Bookmark** property.

To restrict the fields that the **GetRows** method returns, you can pass either a single field name/number or an array of field names/numbers in the *Fields* argument.

After the **GetRows** method is called, the next unread record becomes the current record, or the **EOF** property is set to **True** if there are no more records.

IsolationLevel Property

The **IsolationLevel** property on a **Connection** object indicates the level of isolation for a Connection object. This property sets or returns a Long value representing an **IsolationLevelEnum**. The default value for this property is **adXactChaos.**

```
currentIsoLevel = connection.IsolationLevel
connection.IsolationLevel = newIsoLevel
```

Remarks

The **IsolationLevel** property is used to set the type of isolation level placed on a connection that the provider should use when opening the **Conection** object. This property can also be used to return the type of isolation level in use on an open **Connection** object.

This property can be one of the following enumerated values for **IsolationLevelEnum**:

Enumeration	Value	Description
adXactUnspecified	-1	This value indicates that the provider is using a different isolation level than specified, but that the level cannot be determined.
adXactChaos	16	This value indicates that pending changes from more highly isolated transactions cannot be overwritten.
adXactBrowse	256	This value indicates that from one transaction you can view uncommitted changes in other transactions.
adXactReadUncommitted	256	Same as **adXactBrowse**.
adXactCursorStability	4096	This value indicates that from one transaction you can view changes in other transactions only after they have been committed.
adXactReadCommitted	4096	Same as **adXactCursorStability**.
adXactRepeatableRead	65536	This value indicates that from one transaction you cannot see changes made in other transactions, but that requerying can retrieve new **Recordset** objects.
adXactIsolated	1048576	This value indicates that transactions are conducted in isolation of other transactions.
adXactSerializable	1048576	Same as **adXactIsolated**.

This property is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM.

When setting the **IsolationLevel** property, this change does not take effect until the next time that the **BeginTrans** method is called.

If the level of isolation you request is unavailable, the provider may return the next greater level of isolation.

When used with the Microsoft® OLE DB Provider for DB2 or the Microsoft® OLE DB Driver for DB2, the ADO IsolationLevel property corresponds with the isolation level defined by the ANSI SQL standard and with IBM's DB2 implementation of isolation level. The table below indicates how the ADO IsolationLevel Property corresponds with the terms used by ANSI SQL for isolation level and with IBM documentation for isolation level in DB2.

ADO IsolationLevel Property	ANSI SQL Isolation Level	IBM Documentation
	AUTOCOMMITTED (Note that this applies only to DB2/400 and does not correspond with an ANSI SQL isolation level	COMMIT(*NONE) (NC). This isolation level is used in DB2/400 auto-commit mode only and has no corresponding isolation level on other DB2 platforms or in ANSI SQL.
adXactReadUncommitted	READ UNCOMMITTED	UNCOMMITTED READ (UR). This isolation level corresponds with ANSI SQL READ UNCOMMITTED.

adXactReadCommitted or **adXactCursorStability**	READ COMMITTED	CURSOR STABILITY (CS). This isolation level corresponds with ANSI SQL READ COMMITTED.
adXactRepeatableRead	REPEATABLE READ	READ STABILITY (RS). This isolation level corresponds with ANSI SQL REPEATABLE READ.
adXactSerializable or **adXactIsolated**	SERIALIZABLE	REPEATABLE READ (RR). This isolation level corresponds with ANSI SQL SERIALIZABLE.

When used with the Remote Data Service on a client-side **Connection** object, the **IsolationLevel** property can be set only to **adXactUnspecified**. Because users are working with disconnected **Recordset** objects on a client-side cache, there may be multiuser issues. For instance, when two different users try to update the same record, Remote Data Service simply allows the user who updates the record first to "win." The second user's update request will fail with an error.

Item Method

The **Item** method on a **Collection** object returns a specific member object of a collection by name or ordinal number. This method is supported for the **Errors**, **Field**s, and **Properties** collections using the OLE DB Provider for AS/400 and VSAM.

```
set Object = collection.Item ( Index )
```

Parameters

Index
This parameter specifies a Variant that evaluates either to the name or to the ordinal number of an object in a collection.

Return Values

An object reference from the collection.

Remarks

The **Item** method is used to return a specific object in a collection. If the method cannot find an object in the collection corresponding to the *Index* parameter, an error occurs. Also, some collections do not support named objects; for these collections, you must use ordinal number references.

The **Item** method is the default method for all collections; therefore, the following syntax forms are interchangeable:

collection.Item (*Index*)

collection (*Index*)

This method is only supported on the **Errors**, **Field**s, and **Properties** collections under the OLE DB Provider for AS/400 and VSAM.

LockType Property

The **LockType** property on a **Recordset** object indicates the type of locks placed on records during editing. This property sets or returns a Long value representing a **LockTypeEnum**. The default value for this property is **adLockReadOnly**.

```
currentLock = recordset.LockType
recordset.LockType = newtype
```

Remarks

The **LockType** property is used to set the type of locks placed on records that the provider should use when opening the **Recordset**. This property can also be used to return the type of locking in use on an open **Recordset** object. This property can be one of the following enumerated values for **LockTypeEnum**:

Enumeration	Value	Description
adLockUnspecified	-1	This value does not specify a type of lock. For a recordset created with the **Clone** method, the clone is created with the same lock type as the original.
adLockReadOnly	1	This value indicates read-only records where the data cannot be altered.
adLockPessimistic	2	This value indicates pessimistic locking, record by record. The provider does what is necessary to ensure successful editing of the records, usually by locking records at the data source immediately after editing. This lock type is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2. However, the OLE DB Provider for AS/400 and VSAM internally maps this lock type to **adLockBatchOptimistic**.
adLockOptimistic	3	This value indicates optimistic locking, record by record. The provider uses optimistic locking, locking records only when the **Update** method is called. This lock type is not supported by the OLE DB Provider for DB2.
adLockBatchOptimistic	4	This value indicates optimistic batch updates and is required for batch update mode. This option is not supported by the OLE DB Provider for DB2.

If a provider cannot support the requested **LockType** setting, it will substitute another type of locking. To determine the actual locking options available on a **Recordset** object, use the **Supports** method with **adUpdate** and **adUpdateBatch**.

The **adLockPessimistic** setting is not supported if the **CursorLocation** property is set to **adUseClient**. If an unsupported value is set, then no error will result; the closest supported **LockType** will be used instead.

The **LockType** property is read/write when the **Recordset** is closed and read-only when it is open.

MaxRecords Property

The **MaxRecords** property on a **Recordset** indicates the maximum number of records to return to a **Recordset** from a query. This property sets or returns a Long value that indicates the maximum number of records to return. The default value for this property is zero (no limit).

```
cntRecords = recordset.MaxRecords
recordset.MaxRecords = count
```

Remarks

The **MaxRecords** property is used to limit the number of records returned from a data source. The default setting of this property is zero, which means that all requested records are returned.

The **MaxRecords** property is read/write when the **Recordset** is closed and read-only when it is open.

Mode Property

The **Mode** property on a **Connection** object sets or returns the available permissions for modifying data on a **Connection**.

```
currentMode = connection.Mode
connection.Mode = newMode
```

Remarks

The **Mode** property is used to set or return the access permissions in use by the provider on the current connection. The Mode property can only be set when the **Connection** object is closed.

Several of the ADO enumerated values for the mode settings imply that the using certain mode settings will prevent other applications from opening a connection to the file or table. Under the Microsoft® OLE DB provider for AS/400 and VSAM, these mode settings result in a file lock, but don't prevent other applications from opening a connection.

The value for the **Mode** can be one of the following enumerated values for **ConnectModeEnum**:

Enumeration	Value	Description
adModeUnknown	0	This value indicates that the permissions have not yet been set or cannot be determined.
adModeRead	1	This value indicates read-only permissions.
adModeWrite	2	This value indicates write-only permissions. This value is not supported by the OLE DB provider for AS/400 and VSAM.
adModeReadWrite	3	This value indicates read/write permissions.
adModeShareDenyRead	4	This value prevents others from opening a connection with read permissions.
adModeShareDenyWrite	8	This value prevents others from opening a connection with write permissions. Under the OLE DB provider for AS/400 and VSAM, this mode setting is implemented as a file lock and doesn't result in excluding other applications from opening a connection. Other applications opening a connection will not receive an error, but the table or file will be locked preventing any changes from other applications.
adModeShareExclusive	0xc	This value prevents others from opening a connection. Under the OLE DB provider for AS/400 and VSAM, this mode setting is implemented as a file lock and doesn't result in excluding other applications from opening a connection. Other applications opening a connection will not receive an error, but the table or file will be locked preventing any changes from other applications.
adModeShareDenyNone	0x10	This value prevents others from opening a connection with any permissions. Under the OLE DB provider for AS/400 and VSAM this mode setting is implemented as a file lock and doesn't result in excluding other applications from opening a connection. Other applications opening a connection will not receive an error, but the table or file will be locked preventing any changes from other applications.

The **Mode** property defaults to **adModeUnknown**.

Move Method

The **Move** method on a **Recordset** object moves the position of the current record in a **Recordset** object.

```
recordset.Move NumRecords, Start
```

Parameters

NumRecords
This parameter specifies a signed Long expression specifying the number of records the current record position moves.

Start
This optional parameter specifies the starting location for the record from which the **Move** operation should begin, which can be a bookmark or an enumeration indicating the current, first, or last record in a **Recordset** object. This argument is a Variant.

It can be either a bookmark or one of the following enumerated values for **BookmarkEnum**:

Enumeration	Value	Description
adBookmarkCurrent	0	The current record
adBookmarkFirst	1	The first record
adBookmarkLast	2	The last record

This optional argument defaults to **adBookmarkCurrent**.

Return Values

None.

Remarks

The **Move** method is supported on all **Recordset** objects.

If the *NumRecords* parameter is greater than zero, the current record position moves forward (toward the end of the recordset). If *NumRecords* is less than zero, the current record position moves backward (toward the beginning of the recordset).

If the **Move** method would move the current record position to a point before the first record, ADO sets the current record to the position before the first record in the recordset (the **BOF** property is set to **True**). An attempt to move backward when the **BOF** property is already **True** generates an error.

If the **Move** call would move the current record position to a point after the last record, ADO sets the current record to the position after the last record in the recordset (the **EOF** property is set to **True**). An attempt to move forward when the **EOF** property is already **True** generates an error.

Invoking the **Move** method from an empty **Recordset** object generates an error.

If the *Start* parameter is specified, the move is relative to the record with this bookmark assuming the **Recordset** object supports bookmarks. If not specified, the move is relative to the current record.

If the **CacheSize** property is set to greater than 1 to locally cache records from the provider, passing a *NumRecords* value that moves the current record position outside of the current group of cached records forces ADO to retrieve a new group of records starting from the destination record. The **CacheSize** property determines the size of the newly retrieved group, and the destination record is the first record retrieved.

If the **Recordset** object is forward-only, a user can still pass a *NumRecords* value less than zero as long as the destination is within the current set of cached records. If the **Move** method call would move the current record position to a record before the first cached record, an error occurs. Thus, you can use a record cache that supports full scrolling over a provider that only supports forward scrolling. Because cached records are loaded into memory, you should avoid caching more records than necessary. Even if a forward-only **Recordset** object supports backward moves in this way, calling the **MovePrevious** method on any forward-only **Recordset** object still generates an error.

MoveFirst Method

The **MoveFirst** method on a **Recordset** object moves to the first record in a specified **Recordset** object and makes that record current.

```
recordset.MoveFirst
```

Parameters

None.

Remarks

The **MoveFirst** method is used to move the current record position to the first record in the recordset. The **MoveFirst** method can be invoked on a forward-only **Recordset** object; but doing so may cause the provider to re-execute the command that generated the **Recordset** object.

MoveLast Method

The **MoveLast** method on a **Recordset** object moves to the last record in a specified **Recordset** object and makes that record current.

```
recordset.MoveLast
```

Parameters

None.

Remarks

The **MoveLast** method is used to move the current record position to the last record in the recordset.

When using a server-side cursor, the **Recordset** object must support bookmarks or backward cursor movement; otherwise, the **MoveLast** method call generates an error.

MoveNext Method

The **MoveNext** method on a **Recordset** object moves to the next record in a specified **Recordset** object and makes that record current.

```
recordset.MoveLast
```

Parameters

None.

Remarks

The **MoveNext** method is used to to move the current record position one record forward (toward the bottom of the recordset). If the last record is the current record and the **MoveNext** method is invoked, ADO sets the current record to the position after the last record in the recordset (the **EOF** property is set to **True**). An attempt to move forward when the **EOF** property is already **True** generates an error.

MovePrevious Method

The **MovePrevious** method on a **Recordset** object moves to the previous record in a specified **Recordset** object and makes that record current.

```
recordset.MovePrevious
```

Parameters

None.

Remarks

The **MovePrevious** method is used to move the current record position one record backward (toward the top of the recordset).

When using a server-side cursor, if the **Recordset** object does not support either bookmarks or backward cursor movement, the **MovePrevious** method generates an error.

If the first record is the current record and the **MovePrevious** method is invoked, ADO sets the current record to the position before the first record in the recordset (the **BOF** property is set to **True**). An attempt to move backward when the **BOF** property is already **True** generates an error.

Note that if a **MovePrevious** method is invoked after **Delete** method is invoked, this moves the current record back two records instead of one.

Name Property

The **Name** property on a **Command** object sets or returns a string value indicating the name of the object. The **Name** property on a **Field** or **Property** returns a string value indicating the name of the object.

```
currentName = command.Name
command.Name = newName
```

Remarks

The **Name** property is used to assign a name to or retrieve the name of a **Command**, **Field**, or **Property** object. This value is read/write on a **Command** object and read-only on a **Property** or **Field** object. Note that the **Name** property on a **Command** object cannot be assigned (set) using the OLE DB Provider for AS/400 and VSAM.

The **Name** property of an object can be retrieved by an ordinal reference, after which you can refer to the object directly by name. For example, if recordset.Properties(20).Name yields Updatability, you can subsequently refer to this property as recordset.Properties("Updatability").

NativeError Property

The **NativeError** property on a **Error** object indicates the provider-specific error code for a given **Error** object. This property returns a a Long value that indicates the error code.

```
errorCode = currentError.NativeError
```

Remarks

The **NativeError** property on a **Error** object is used to retrieve the database-specific error information for a particular **Error** object. For example, when using the Microsoft OLE DB Provider for DB2, native error codes that originate from the OLE DB Provider for DB2 pass through ADO to the ADO **NativeError** property.

Number Property

The **Number** property on a **Error** object indicates the number that uniquely identifies an **Error** object. This property returns a a Long value that may correspond to one of the **ErrorValueEnum** constants.

```
errorNumber = currentError.Number
```

Remarks

The **Number** property on a **Error** object is used to determine which error occurred. The value of the property is a unique number that corresponds to the error condition.

The **Errors** collection returns an HRESULT. These HRESULTs can be raised by underlying components, such as OLE DB or even OLE itself.

The value for the **Number** property on the **Error** object representing an ADO error (not an OLE DB error) can be one of the following enumerated values for **ErrorValueEnum** listed in the table below. Note that three forms of the error number value are listed in the table:

- Positive decimal—The lower two bytes of the full number in decimal format. This number is displayed in the default Visual Basic error message dialog box. For example, **Run-time error '3707'**.
- Negative decimal—The decimal translation of the full error number.
- Hexadecimal—The hexadecimal representation of the full error number. The Windows facility code is the lowest hexadecimal fourth digit in the upper two bytes of the number. The facility code for ADO error numbers is *A*. For example: 0x800**A**0E7B.

Note that OLE DB errors may be passed to an ADO application. Typically, these OLE DB errors can be identified by a Windows facility code of *4*. For example, 0x800**4**xxxx. For more information about the possible error numbers returned by OLE DB, see Chapter 16 of the *OLE DB Programmer's Reference*.

Enumeration	Value	Description
adErrBoundToCommand	3707 -2146824581 0x800A0E7B	The application cannot change the **ActiveConnection** property of a **Recordset** object that has a **Command** object as its source.
adErrCannotComplete	3732 -2146824556 0x800A0E94	The server cannot complete the operation.
adErrCantChangeConnection	3748 -2146824540 0x800A0EA4	The connection was denied. The new connection requested has different characteristics than the one already in use.
adErrCantChangeProvider	3220 -2146825068 0X800A0C94	The supplied provider is different from the one already in use.
adErrCantConvertvalue	3724 -2146824564 0x800A0E8C	The data value cannot be converted for reasons other than sign mismatch or data overflow. For example, conversion would have truncated data.
adErrCantCreate	3725 -2146824563 0x800A0E8D	The data value cannot be set or retrieved because the field data type was unknown, or the provider had insufficient resources to perform the operation.

adErrCatalogNotSet	3747 -214682 4541 0x800A 0EA3	The operation requires a valid **ParentCatalog**.
adErrColumnNotOnThisRow	3726 -214682 4562 0x800A 0E8E	The record does not contain this field.
adErrDataConversion	3421 -214682 4867 0x800A 0D5D	The application uses a value of the wrong type for the current operation.
adErrDataOverflow	3721 -214682 4567 0x800A 0E89	The data value is too large to be represented by the field data type.
adErrDelResOutOfScope	3738 -214682 4550 0x800A 0E9A	The URL of the object to be deleted is outside the scope of the current record.
adErrDenyNotSupported	3750 -214682 4538 0x800A 0EA6	The provider does not support sharing restrictions.
adErrDenyTypeNotSupported	3751 -214682 4537 0x800A 0EA7	The provider does not support the requested kind of sharing restriction.
adErrFeatureNotAvailable	3251 -214682 5037 0x800A 0CB3	The object or provider is not capable of performing requested operation.
adErrFieldsUpdateFailed	3749 -214682 4539 0x800A 0EA5	The **Fields** update failed. For further information, examine the **Status** property of individual field objects.
adErrIllegalOperation	3219 -214682 5069 0x800A 0C93	The operation is not allowed in this context.
adErrIntegrityViolation	3719 -214682 4569 0x800A 0E87	The data value conflicts with the integrity constraints of the field.
adErrInTransaction	3246 -214682 5042 0x800A 0CAE	The **Connection** object cannot be explicitly closed while in a transaction.

adErrInvalidArgument	3001 -2146825287 0x800A0BB9	The arguments are of the wrong type, are out of acceptable range, or are in conflict with one another.
adErrInvalidConnection	3709 -2146824579 0x800A0E7D	The operation is not allowed on an object referencing a closed or invalid connection.
adErrInvalidParamInfo	3708 -2146824580 0x800A0E7C	The **Parameter** object is improperly defined. Inconsistent or incomplete information was provided.
adErrInvalidTransaction	3714 -2146824574 0x800A0E82	The coordinating transaction is invalid or has not started.
adErrInvalidURL	3729 -2146824559 0x800A0E91	The URL contains invalid characters. Make sure the URL is typed correctly.
adErrItemNotFound	3265 -2146825023 0x800A0CC1	The item cannot be found in the collection corresponding to the requested name or ordinal.
adErrNoCurrentRecord	3021 -2146825267 0x800A0BCD	Either the **BOF** or **EOF** property is True, or the current record has been deleted. The requested operation requires a current record.
adErrNotExecuting	3715 -2146824573 0x800A0E83	The operation cannot be performed while not executing.
adErrNotReentrant	3710 -2146824578 0x800A0E7E	The operation cannot be performed while processing event.
adErrObjectClosed	3704 -2146824584 0x800A0E78	The operation is not allowed when the object is closed.
adErrObjectInCollection	3367 -2146824921 0x800A0D27	The object is already in collection. Cannot append.
adErrObjectNotSet	3420 -2146824868 0x800A0D5C	The object is no longer valid.

adErrObjectOpen	3705 -2146824583 0x800A0E79	The operation is not allowed when the object is open.
adErrOpeningFile	3002 -2146825286 0x800A0BBA	The file could not be opened.
adErrOperationCancelled	3712 -2146824576 0x800A0E80	The operation has been cancelled by the user.
adErrOutOfSpace	3734 -2146824554 0x800A0E96	The operation cannot be performed. Provider cannot obtain enough storage space.
adErrPermissionDenied	3720 -2146824568 0x800A0E88	Insufficent permission prevents writing to the field.
adErrPropConflicting	3742 -2146824546 0x800A0E9E	Property value conflicts with a related property.
adErrPropInvalidColumn	3739 -2146824549 0x800A0E9B	Property cannot apply to the specified field.
adErrPropInvalidOption	3740 -2146824548 0x800A0E9C	Property attribute is invalid.
adErrPropInvalidValue	3741 -2146824547 0x800A0E9D	Property value is invalid. Make sure the value is typed correctly.
adErrPropNotAllSettable	3743 -2146824545 0x800A0E9F	Property is read-only or cannot be set.
adErrPropNotSet	3744 -2146824544 0x800A0EA0	Optional property value was not set.
adErrPropNotSettable	3745 -2146824543 0x800A0EA1	Read-only property value was not set.

adErrPropNotSupported	3746 -2146824542 0x800A0EA2	Provider does not support the property.
adErrProviderFailed	3000 -2146825288 0x800A0BB8	Provider failed to perform the requested operation.
adErrProviderNotFound	3706 -2146824582 0x800A0E7A	Provider cannot be found. It may not be properly installed.
adErrReadFile	3003 -2146825285 0x800A0BBB	File could not be read.
adErrResourceExists	3731 -2146824557 0x800A0E93	Copy operation cannot be performed. Object named by destination URL already exists. Specify **adCopyOverwrite** to replace the object.
adErrResourceLocked	3730 -2146824558 0x800A0E92	Object represented by the specified URL is locked by one or more other processes. Wait until the process has finished and attempt the operation again.
adErrResourceOutOfScope	3735 -2146824553 0x800A0E97	Source or destination URL is outside the scope of the current record.
adErrSchemaViolation	3722 -2146824566 0x800A0E8A	Data value conflicts with the data type or constraints of the field.
adErrSignMismatch	3723 -2146824565 0x800A0E8B	Conversion failed because the data value was signed and the field data type used by the provider was unsigned.
adErrStillConnecting	3713 -2146824575 0x800A0E81	Operation cannot be performed while connecting asynchronously.
adErrStillExecuting	3711 -2146824577 0x800A0E7F	Operation cannot be performed while executing asynchronously.
adErrTreePermissionDenied	3728 -2146824560 0x800A0E90	Permissions are insufficient to access tree or subtree.

adErrUnavailable	3736 -214682 4552 0x800A 0E98	Operation failed to complete and the status is unavailable. The field may be unavailable or the op eration was not attempted.
adErrUnsafeOperatio n	3716 -214682 4572 0x800A 0E84	Safety settings on this computer prohibit accessing a data source on another domain.
adErrURLDoesNotExis t	3727 -214682 4561 0x800A 0E8F	Either the source URL or the parent of the destination URL does not exist.
adErrURLNamedRow DoesNotExist	3737 -214682 4551 0x800A 0E99	Record named by this URL does not exist.
adErrVolumeNotFoun d	3733 -214682 4555 0x800A 0E95	Provider cannot locate the storage device indicated by the URL. Make sure the URL is typed corre ctly.
adErrWriteFile	3004 -214682 5284 0x800A 0BBC	Write to file failed.

NumericScale Property

The **NumericScale** property on a **Field** object indicates the scale of Numeric values in a **Field** object. This property returns a byte value indicating the number of decimal places to which numeric values will be resolved.

```
scale = currentfield.NumericScale
```

Remarks

The **NumericScale** property is used to determine how many digits to the right of the decimal point will be used to represent values for a numeric **Field** object.

The byte value that the **NumericScale** property will return is dependent on the data type of the **Field** object. The value for the ADO data type of the **Field** object can be one of the following enumerated values for **DataTypeEnum**:

Enumeration	Value	Description
adEmpty	0	This data type indicates that no value was specified (DBTYPE_EMPTY).
adSmallInt	2	This data type indicates a two-byte (16-bit) signed integer (DBTYPE_I2).
adInteger	3	This data type indicates a four-byte (32-bit) signed integer (DBTYPE_I4).
adSingle	4	This data type indicates a four-byte (32-bit) single precision IEEE floating point number (DBTYPE_R4).
adDouble	5	This data type indicates an eight-byte (64-bit) double precision IEEE floating point number (DBTYPE_R8).
adCurrency	6	A data type indicates a currency value (DBTYPE_CY). Currency is a fixed-point number with 4 digits to the right of the decimal point. It is stored in an eight-byte signed integer scaled by 10,000. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDate	7	This data type indicates a date value stored as a Double, the whole part of which is the number of days since December 30, 1899, and the fractional part of which is the fraction of a day. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBSTR	8	This data type indicates a null-terminated Unicode character string (DBTYPE_BSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIDispatch	9	This data type indicates a pointer to an IDispatch interface on an OLE object (DBTYPE_IDISPATCH). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adError	10	This data type indicates a 32-bit error code (DBTYPE_ERROR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBoolean	11	This data type indicates a Boolean value (DBTYPE_BOOL). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adVariant	12	This data type indicates an automation variant (DBTYPE_VARIANT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIUnknown	13	This data type indicates a pointer to an IUnknown interface on an OLE object (DBTYPE_IUNKNOWN). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDecimal	14	This data type indicates numeric data with a fixed precision and scale (DBTYPE_DECIMAL).
adTinyInt	16	This data type indicates a single -byte (8-bit) signed integer (DBTYPE_I1). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adUnsignedTinyInt	17	This data type indicates a single-byte (8-bit) unsigned integer (DBTYPE_UI1). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adUnsignedSmallInt	18	This data type indicates a two-byte (16-bit) unsigned integer (DBTYPE_UI2). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

adUnsignedInt	19	This data type indicates a four-byte (32-bit) unsigned integer (DBTYPE_UI4). This data type is not supported by the OLE DB Provider or the OLE DB Provider for DB2.
adBigInt	20	This data type indicates an eight-byte (64-bit) signed integer (DBTYPE_I8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM.
adUnsignedBigInt	21	This data type indicates an eight-byte (64-bit) unsigned integer (DBTYPE_UI8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adGUID	72	This data type indicates a globally unique identifier or GUID (DBTYPE_GUID). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBinary	128	This data type indicates fixed-length binary data (DBTYPE_BYTES).
adChar	129	This data type indicates a character string value (DBTYPE_STR).
adWChar	130	This data type indicates a null-terminated Unicode character string (DBTYPE_WSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adNumeric	131	This data type indicates numeric data where the precision and scale are exactly as specified (DBTYPE_NUMERIC).
adUserDefined	132	This data type indicates user-defined data (DBTYPE_UDT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDBDate	133	This data type indicates a OLE DB date structure (DBTYPE_DATE).
adDBTime	134	This data type indicates a OLE DB time structure (DBTYPE_TIME).
adDBTimeStamp	135	This data type indicates a OLE DB timestamp structure (DBTYPE_TIMESTAMP).
adVarChar	200	This data type indicates variable-length character data (DBTYPE_STR).
adLongVarChar	201	This data type indicates a long string value.
adVarWChar	202	This data type indicates a Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adLongVarWChar	203	This data type indicates a long Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adVarBinary	204	This data type indicates variable-length binary data (DBTYPE_BYTES).
adLongVarBinary	205	This data type indicates a long binary value.

Open Method on Connection Object

The **Open** method on a **Connection** object opens a connection to a data source.

```
connection.Open ConnectionString, UserID, Password
```

Parameters

ConnectionString
This optional parameter specifies a String containing connection information. See the **ConnectionString** property on a **Connection** object for details on valid settings. Possible values are listed in the table following the Parameters list.

Note Not all of these parameters are required. The user can also be prompted for this information.

UserID
This optional parameter specifies a String containing a user name to use when establishing the connection.
Password
This optional parameter specifies a String containing a password to use when establishing the connection.

Values for the ConnectionString parameter

The information needed to establish a connection to a data source can be set in the **ConnectionString** property or passed as part of the **Open** method in the *ConnectionString* parameter. In either case, this information must be in a specific format for use with the Microsoft® OLE DB Provider for AS/400 and VSAM. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons. ADO supports several standard ADO-defined arguments for the **ConnectionString** property as follows:

Argument	Description
Data Source	This argument specifies the name of the data source for the connection. This argument is the Data Source name stored in the registry under the OLE DB Provider for AS/400 and VSAM.
File Name	This argument specifies the name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed. This argument is not supported by the OLE DB Provider for AS/400 and VSAM.
Location	This argument specifies the Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	This argument specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target host system and has appropriate access rights to the file.
Provider	This argument specifies the name of the provider to use for the connection. To use the OLE DB Provider for AS/400 and VSAM, the Provider string must be set to "SNAOLEDB". To use the Microsoft® OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB". To use the Microsoft® ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
Remote Provider	This argument specifies the name of a provider to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
Remote Server	This argument specifies the path name of a server to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
User ID	This argument specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target host system and has appropriate access rights to the file.

The OLE DB Provider for AS/400 and VSAM also supports a number of provider-specific arguments, some of which have default values as specified in the table below. These arguments are as follows:

Argument	Description
BinAsChar	This parameter indicates whether to process binary fields as character fields (default is 0, don't process binary fields as character fields).
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files.
HCDFileName	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
RDB	The Remote DataBase name for OS/400. You only need to specify this value if it is different from the remote LU alias configured in Host Integration Server.
Repair Host Keys	This parameter indicates whether the OLE DB provider should repair any host key values set in the registry and defaults to false.
RemoteLU	The name of the remote LU alias configured in the Host Integration server.
StrictVal	This parameter indicates whether strict validation should be used and defaults to false.

The OLE DB Provider for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by OLE DB Provider for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier OLE DB Provider for DB2 included with SNA Server 4.0.

The arguments supported by the OLE DB Provider for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	Description
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.

CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDB NAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 syste m. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is ref erred to as DATABASE. This parameter has no default value.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATC HSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this par ameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will c reate packages dynamically in the location to which the user points using the Package Collection parameter.
RemoteLU	The name of the remote LU alias configured in Host Integration Server.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA applicati on server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, T PName is set to 0X07F9F9F9.

UOW	This parameter determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the OLE DB Provider for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
BindType	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Commit	This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter defaults to true (auto commit).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.

GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
IsoLvl	This parameter determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommited Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommited. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
LocalLU	The name of the local LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.

NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
ReadOnly	When the Read Only parameter is set to true (ReadOnly=1), the OLE DB Provider for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RemoteLU	The name of the remote LU alias configured in Host Integration.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.

Not all of these parameters are required. The user can also be prompted for this information.

The ODBC Driver for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by ODBC Driver for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier ODBC Driver for DB2 included with SNA Server 4.0.

The arguments supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	**Description**
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.

CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.

PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPN is set to 0X07F9F9F9.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.
UOW	Determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the ODBC Driver for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**

ACM	The Auto Commit Mode parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true (auto commit). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
BT	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.

DIL	This Default Isolation Level parameter determines the isolation level provided for this data source in cases of simultaneous access to DB2 objects by multiple applications. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Please note that the ALL isolation level is not allowed. Users should set the isolation level to RS since this has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.

NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RO	When the Read Only parameter is set to true (RO=1), the ODBC Driver for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

Remarks

The **Open** method on a **Connection** object is used to open tables on a remote DDM server. Using the **Open** method on a **Connection** object establishes the physical connection to a data source. After this method successfully completes, the connection is live and other methods can be invoked on the **Connection** object to process results.

The optional *ConnectionString* parameter is used to specify a connection string containing a series of *argument=value* statements separated by semicolons. The **ConnectionString** property on a **Connection** object automatically inherits the value used for the *ConnectionString* parameter. Therefore, the **ConnectionString** property of the **Connection** object can be set before opening the **Connection** object, or the *ConnectionString* parameter can be used to set or override the current connection parameters during the **Open** method call.

If user and password information is set in both the *ConnectionString* parameter and in the optional *UserID* and *Password* parameters, the results may be unpredictable. Such information should only be passed in either the *ConnectionString* parameter

or the *UserID* and *Password* parameters.

There are a number of different ways to open a connection. The **Open** method can pass all of the appropriate connection information as part of the *ConnectionString* parameter or by setting the **ConnectionString** property of the **Connection** object, if this information is known in advance. The syntax in this case using the *ConnectionString* parameter for use with the OLE DB Provider for AS/400 and VSAM is as follows:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.Open "Provider=SNAOLEDB;Data Source=REMLU;User ID=USERNAME;Password=password;Local
LU=LOCAL;Remote LU=DATABASE;ModeName=QPCSUPP;CCSID=37;CodePage=437"
```

Note that not all of these parameters are required. The registry settings for the Data Source usually have default values set for remote LU, local LU, APPC mode, CCSID, and CodePage. If a data source is specified, this other information is not usually needed. These registry settings are configured by using the Microsoft Management Console snap-in for the OLE DB Provider for AS/400 and VSAM.

The simplest form of an **Open** command that contains all necessary information is as follows:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.Open "Provider=SNAOLEDB;Data Source=REMLU;User ID=USERNAME;Password=password"
```

The Data Source, User ID and Password must be included.

In the case where you would like the user to input the connection information, the following syntax can be used. This syntax does not specify any connection information except the provider, which is always required unless this is set in the **ConnectionString** or **Provider** property of the **Connection** object:

```
connection = CreateObject("ADODB.Connection.2.0")
connection.ConnectionString = "Provider=SNAOLEDB"
connection.Open
```

This method of invoking the **Open** method automatically causes a dialog box to appear asking the user for the data source, user name, and password.

When operations have been concluded over an open **Connection** object, the **Close** method should be invoked on the **Connection** object to free any associated system resources. Closing a **Connection** object does not remove it from memory; you may change its property settings and use the **Open** method to open it again later. To completely eliminate an object from memory, set the **Connection** object variable to **Nothing**.

If errors occur, these can be examined with the **Errors** collection on the **Connection** object.

Open Method on Recordset Object

The **Open** method on a **Recordset** object opens a cursor that represents records from a base table or the results of a query.

```
recordset.Open Source, ActiveConnection, CursorType, LockType,
Options
```

Parameters

Source
This optional parameter specifies a Variant that evaluates to a valid **Command** object variable name or a valid string specifying the command text specific to the Microsoft® OLE DB Provider for AS/400 and VSAM to open a data file on the host.

ActiveConnection
This optional parameter specifies either a Variant that evaluates to a valid **Connection** object variable name or a String containing connection information equivalent to the **ConnectionString** property of a **Connection** object. Possible values are listed in the table following the **Parameters** section.

CursorType
This optional parameter specifies a **CursorTypeEnum** value that determines the type of cursor that the provider should use when opening the **Recordset**. See the CursorType property of a **Recordset** object for more information. Possible values are listed in the table following the **Parameters** section.

LockType
This optional parameter specifies a **LockTypeEnum** value that determines what type of locking (concurrency) the provider should use when opening the recordset. See the LockType property of a **Recordset** object for more information. Possible values are listed in the table following the **Parameters** section.

Options
This optional parameter specifies a Long value representing a **CommandTypeEnum** value that indicates how the provider should evaluate the *Source* parameter. Possible values are listed in the table following the **Parameters** section.

Possible values for the *ActiveConnection* parameter

The information needed to establish a connection to a data source can be set in the **ActiveConnection** property of a **Recordset** object or passed as part of the **Open** method on a **Recordset** object in the *ActiveConnection* parameter. In either case, this information must be in a specific format for use with the OLE DB Provider for AS/400 and VSAM. This information can be a data source name (DSN) or a detailed connection string containing a series of *argument=value* statements separated by semicolons. ADO supports several standard ADO-defined arguments for the **ActiveConnection** property as follows:

Argument	Description
Data Source	This argument specifies the name of the data source for the connection. This argument is the optional when used with OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE Provider for DB2.
File Name	This argument specifies the name of the provider-specific file containing preset connection information. This argument cannot be used if a *Provider* argument is passed. This argument is not supported by the OLE DB Provider for AS/400 and VSAM.
Location	This argument specifies the Remote Database Name used for connecting to OS/400 systems. This parameter is optional when connecting to mainframe systems.
Password	This argument specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used by Microsoft® SNA Server to validate that the user can log on to the target host system and has appropriate access rights to the file.
Provider	This argument specifies the name of the provider to use for the connection. To use the OLE DB Provider for AS/400 and VSAM, the Provider string must be set to "SNAOLEDB". To use the OLE DB Provider for DB2, the Provider string must be set to "DB2OLEDB". To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
Remote Provider	This argument specifies the name of a provider to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

Remote Server	This argument specifies the path name of a server to use when opening a client-side connection (for a Remote Data Service only). This argument is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
User ID	This argument specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used by SNA Server to validate that the user can log on to the target host system and has appropriate access rights to the file.

The OLE DB Provider for AS/400 and VSAM also supports a number of provider-specific arguments, some of which have default values as specified in the table below. These arguments are as follows:

Argument	**Description**
BinAsChar	This parameter indicates whether to process binary fields as character fields (default is 0, don't process binary fields as character fields).
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Default Library	The default AS/400 library to be accessed. This parameter is not required for mainframe access and is optional when connecting to AS/400 files.
HCD FileName	The fully qualified filename of the DDM host column description (HCD) file. This parameter can be an UNC string up to 256 characters in length. A path does not need to be included in the name if the HCD file is located in the SNA system directory. This parameter is required when connecting to mainframe systems and is optional when connecting to OS/400.
Local LU	The name of the local LU alias configured in the SNA server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and SNA Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
RDB	The Remote DataBase name for OS/400. You only need to specify this value if it is different from the remote LU alias configured in the SNA server.
Repair Host Keys	This parameter indicates whether the OLE DB provider should repair any host key values set in the registry and defaults to false.
RemoteLU	The name of the remote LU alias configured in the SNA server.
StrictVal	This parameter indicates whether strict validation should be used and defaults to false.

Note Not all of these parameters are required. The user can also be prompted for this information.

The OLE DB Provider for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by OLE DB Provider for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier OLE DB Provider for DB2 included with SNA Server 4.0.

The arguments supported by the OLE DB Provider for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	Description
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
LocalLU	The name of the local LU alias configured in Host Integration Server.
ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.

RemoteLU	The name of the remote LU alias configured in Host Integration Server.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPName is set to 0X07F9F9F9.
UOW	This parameter determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the OLE DB Provider for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**
BinAsChar	When this parameter is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host CCSID and PC Code Page values are required input and output parameters. This parameter defaults to false.
BindType	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this argument is omitted, the default value is U.S./Canada (37).
Commit	This parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter defaults to true (auto commit).

DefSch	The name of the default schema (collection/owner) where the system catalogs resides. This parameter can be QSYS2;SYSIBM; SYSTEM; CURLIB; or USERID depending on platform. This parameter does not have a default value.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
InitCat	This parameter is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. In DB2/400, this parameter is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then one can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE. This parameter has no default value.
IsoLvl	This parameter determines the isolation level provided for this data source. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommited Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommited. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, isolation level this corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
LocalLU	The name of the local LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the OLE DB Provider for DB2 supplied with the Japanese version of the OLE DB Provider for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the OLE DB Provider for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.

ModeName	The APPC mode (must be set to a value that matches the host configuration and Host Integration Server configuration). Legal values for the APPC mode include QPCSUPP (5250), #NTER (interactive), #NTERSC (interactive), #BATCH (batch), #BATCHSC (batch), and custom modes.
NetAddr	When TCP/IP has been selected for the Network Transport Library, this parameter indicates the IP address of the host.
NetPort	When TCP/IP has been selected for the Network Transport Library, this parameter is the TCP/IP port used for communication with the source. The default value is TCP/IP port 446.
NetLib	This parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA.
PCCodePage	The character code page to use on the PC. If this argument is omitted, the default value is set to Latin 1 (1252).
PkgCol	The name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2 uses packages to issue dynamic and static SQL statements. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection parameter.
ReadOnly	When the Read Only parameter is set to true (ReadOnly=1), the OLE DB Provider for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RemoteLU	The name of the remote LU alias configured in Host Integration.
TPName	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.

Note Not all of these parameters are required. The user can also be prompted for this information.

The ODBC Driver for DB2 also supports a number of provider-specific arguments, some of which have default values as specified in the tables below. The arguments supported by ODBC Driver for DB2 supplied with Host Integration Server 2000 differ from the arguments supported by the earlier ODBC Driver for DB2 included with SNA Server 4.0.

The arguments supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000 are as follows:

Argument	**Description**

BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.
NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.

PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "MSDASQL" or not used as part of the ConnectionString since this value is the default for ADO.
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name. Host Integration Server 2000 uses the alternate TP name in the off-line demo configuration (DRDADEMO.UDL). In that case, TPN is set to 0X07F9F9F9.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.
UOW	Determines whether two-phase commit is enabled. The possible values for this parameter are DUW (distributed unit of work) or RUW (remote unit of work). This value defaults to RUW. When this parameter is set to RUW, two-phase commit is disabled. When this parameter is set to DUW, two-phase commit is enabled in the OLE DB Provider for DB2. Distributed transactions are handled using Microsoft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB2 for OS/390 v5R1 or later. This option also requires that SNA (LU 6.2) service is selected as the network transport and Microsoft Transaction Server (MTS) is installed.

Note Not all of these parameters are required. The user can also be prompted for this information.

The arguments supported by the ODBC Driver for DB2 supplied with SNA Server 4.0 are as follows:

Argument	**Description**

ACM	The Auto Commit Mode parameter indicates whether changes to data will be automatically committed or require a separate manual commit request. This parameter allows for implicit COMMIT on all SQL statements. In auto-commit mode, every database operation is a transaction that is committed when performed. This mode is suitable for common transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. No ROLLBACK is allowed when using Auto Commit mode. The default value for this parameter is true (auto commit). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
BAC	When the BinAsChar parameter is set to true (1), the ODBC Driver for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The CCSID and PCCodePage values are required input parameters.
BT	This parameter indicates the bind type to be used when creating packages. Legal values for the package binding type are as follows. NORM—normal binding. FAST—create all 64 package sections optimally in a single network flow. NOSP—reserved for future use and currently not supported. The default value for this parameter is NORM. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
CCSID	The character code set identifier (CCSID) matching the DB2 data as represented on the remote computer. The CCSID property is required when processing binary data as character data. Unless the BinAsChar value is set, character data is converted based on the DB2 column CCSID and default ANSI code page. If this argument is omitted, this parameter defaults to U.S./Canada (37).
CP	The character code page to use on the PC. This parameter is required when processing binary data as character data. Unless the Binary as Character (BAC) value is set, character data is converted based on the default ANSI code page configured in Windows. If this argument is omitted, the default value is set to Latin 1 (1252).
DESC	A field to provide a comment describing this ODBC data source. The description is an optional parameter and may be left blank.
DS	The Default Schema parameter is the name of the Collection where the ODBC Driver for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The ODBC driver uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection (e.g., ODBC Catalog SQLTables). For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, then the ODBC driver uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases, therefore it is essential that the Default Schema value in the data source be defined.
DSN	The data source name is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.

DIL	This Default Isolation Level parameter determines the isolation level provided for this data source in cases of simultaneous access to DB2 objects by multiple applications. Legal values for the default isolation level are the following: CS—Cursor Stability. In DB2/400, this isolation level corresponds to COMMIT(*CS). In ANSI, this isolation level corresponds to Read Committed (RC). NC—No Commit. In DB2/400, this isolation level corresponds to COMMIT(*NONE). In ANSI, this isolation level corresponds to No Commit (NC). UR—Uncommitted Read. In DB2/400, this isolation level corresponds to COMMIT(*CHG). In ANSI, this isolation level corresponds to Read Uncommitted. RS—Read Stability. In DB2/400, this isolation level corresponds to COMMIT(*ALL). In ANSI, this isolation level corresponds to Repeatable Read. RR—Repeatable Read. In DB2/400, this isolation level corresponds to COMMIT(*RR). In ANSI, this isolation level corresponds to Serializable (Isolated). This parameter defaults to NC. Please note that the ALL isolation level is not allowed. Users should set the isolation level to RS since this has the equivalent meaning and is defined in DB2 (ALL is not defined in any DB2 system). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
GCCSID	The graphics character code set identifier (GCCSID) matching the DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for GCCSID are supported by the OLE DB Provider for DB2: 300, 834, 835, 837, or 4396. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
LLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured in Host Integration Server.
MCCSID	The mixed character code set identifier (MCCSID) matching DB2 character data as represented on the remote host computer. This parameter is required when accessing DB2 databases configured to support mixed single-byte (SBCS) and double-byte (DBCS) data. This parameter only applies when accessing DB2 for OS/390 or DB2 for MVS. The following values for MCCSID are supported by the OLE DB Provider for DB2: 930, 931, 933, 935, 937, 939, 5026, or 5035. This parameter defaults to 0 indicating that mixed CCSID conversions are not supported. This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000. This parameter is supported by the Japanese version of the ODBC Driver for DB2 client included with SNA Server 4.0 with Service Pack 2, and by the ODBC Driver for DB2 client included with all versions of SNA Server 4.0 with Service Pack 3 or later.
MN	When SNA is used for the Network Transport Library (NTL), the Mode Name field is the APPC mode and must be set to a value that matches the host configuration and Host Integration Server configuration. Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bidirectional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This parameter normally defaults to QPCSUPP.
NA	When TCP/IP is used for the Network Transport Library (NTL), the Network Address parameter indicates the IP address or the hostname alias of the host DB2 server.
NP	When TCP/IP is used for the Network Transport Library (NTL), the Network Port parameter indicates the TCP/IP port used for communication with the target DB2 DRDA service. The default value is TCP/IP port 446.

NTL	The Network Transport Library parameter determines whether TCP/IP or SNA APPC is used for network communication. The possible values for this parameter are TCPIP or SNA. This value defaults to SNA. If the default SNA is selected, then values for LLU, MN, and RLU are required. If TCP/IP is selected, then values for NetAddr and NetPort are required.
PC	The name of the DRDA target collection (AS/400 library) where the Microsoft ODBC Driver for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft ODBC Driver for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. The ODBC driver will create packages dynamically in the location to which the user points using the Package Collection parameter.
PDS	The Provider Data Source is a required parameter that is used to define the data source. The ODBC driver manager uses this attribute value to load the correct ODBC data source configuration from the registry or from a file. For File data sources, this field is used to name the DSN file which is stored in the Program Files\Common Files\ODBC\Data Sources directory.
PROV	Specifies the name of the provider to use for the connection. To use the ODBC Driver for DB2, the Provider string must be set to "DB2OLEDB".
PWD	Specifies a valid mainframe or AS/400 password to use when opening the connection. This password is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. Note that this parameter is the same as the Parameter parameter.
RDB	The Remote Database Name parameter is used as the first part of a three-part, fully qualified DB2 table name. This parameter is referred to by different names depending on the DB2 platform. In DB2 on MVS and OS/390, this parameter is referred to as LOCATION. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to connect on these platforms, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400 on OS/400, this property is referred to as RDBNAM. The RDBNAM value can be determined by invoking the WRKRDBDIRE command from the console to the OS/400 system. If there is no RDBNAM value, then a value can be created using the Add option. In DB2 Universal Database, this property is referred to as DATABASE.
RO	When the Read Only parameter is set to true (RO=1), the ODBC Driver for DB2 creates a read-only data source. A user has read access to objects such as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This parameter is not supported by the ODBC Driver for DB2 supplied with Host Integration Server 2000.
RLU	When SNA is used for the network transport, this field is the name of the remote LU alias configured inHost Integration Server.
TPN	The Transaction Program (TP) Name parameter represents the default transaction program name for the DB2 DRDA application server (AS) which is 07F6DB (DB2DRDA). However, some DB2 installations may be configured to use an alternate TP name.
UID	Specifies a valid mainframe or AS/400 user name to use when opening the connection. This user name is used to validate that the user can log on to the target DB2 host system and has appropriate access rights to the database. This parameter is the same as the User ID parameter.

Note Not all of these parameters are required. The user can also be prompted for this information.

Possible values for the *LockType* parameter

This parameter can be one of the following enumerated values for **LockTypeEnum**:

Enumeration	Value	Description
adLockUnspecified	-1	Indicates an unspecified value for the *LockType*. This value is not supported by the OLE DB Provider for AS/400 and VSAM.

adLockReadOnly	1	Specifying this value opens a **Recordset** object read-only and data cannot be altered.
adLockPessimistic	2	Specifying this value opens a recordset with pessimistic locking. Record-by-record, the OLE DB Provider does whatever is necessary to ensure successful editing of the records, usually by locking records at the data source immediately upon editing. This lock type is supported by the OLE DB Provider for AS/400 and VSAM and the OLE DB Provider. However, the OLE DB Provider for AS/400 and VSAM internally maps this lock type to **adLockBatchOptimistic**.
adLockOptimistic	3	Specifying this value opens a recordset with optimistic locking. Record-by-record, the OLE DB Provider locks records only when the **Update** method is invoked on a **Recordset** object. This lock type is not supported by the OLE DB Provider for DB2.
adLockBatchOptimistic	4	Specifying this value opens a **Recordset** with batch optimistic locking. This value is required for batch update mode as opposed to immediate update. This lock type is not supported by the OLE DB Provider for DB2.

This optional argument defaults to **adLockReadOnly**.

Possible values for the *CursorType* parameter

This parameter can be one of the following enumerated values for **CursorTypeEnum**:

Enumeration	Value	Description
adOpenUnspecified	-1	This indicates an unspecified value for the *CursorType*. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
adOpenForwardOnly	0	Specifying this value opens a forward-only-type cursor. This *CursorType* is identical to a static cursor, except that you can only scroll forward through records. This improves performance when only one pass through a **Recordset** is needed. This value is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM.
adOpenKeyset	1	Specifying this value opens a keyset-type cursor. This *CursorType is* similar to a dynamic cursor with a few exceptions. Records that other users delete are inaccessible from your **Recordset**. Data changes to existing records by other users are still visible, but records added by other users are not visible (cannot be seen). This value is not supported by the OLE DB Provider for AS/400 and VSAM..
adOpenDynamic	2	Specifying this value opens a dynamic-type cursor. Additions, changes, and deletions by other users are visible, and all types of movement through the recordset are allowed, except for bookmarks if the provider does not support them. A dynamic cursor is the only *CursorType* supported by the OLE DB Provider for AS/400 and VSAM.
adOpenStatic	3	Specifying this value opens a static-type cursor. A static cursor provides a static copy of a set of records that can be used to find data or generate reports. Additions, changes, or deletions by other users are not visible with a static cursor. This value is not supported by the OLE DB Provider for AS/400 and VSAM.

This optional argument defaults to **adOpenForwardOnly**, a value that is mapped to **adOpenDynamic** by the OLE DB provider for AS/400 and VSAM.

Possible values for the *Options* parameter

The **CommandTypeEnum** value can be one of the following constants:

Enumeration	Value	Description

adCmdUnspecified	-1	This value indicates that the **CommandText** property is unspecified. This value is not supported by the OLE DB Provider for AS/400 and VSAM.
adCmdText	1	This value evaluates the **CommandText** property as a textual definition of a command or stored procedure call.
adCmdTable	2	This value evaluates the **CommandText** property as a table name. This value is not supported by the OLE DB Provider for AS/400 and VSAM.
adCmdStoredProc	4	This value evaluates the **CommandText** property as a stored procedure name. This value is not supported by the OLE DB Provider for AS/400 and VSAM.
adCmdUnknown	8	This value indicates that the type of command in **CommandText** property is not known. This is the default value. This value is not supported by the OLE DB Provider for AS/400 and VSAM.

Remarks

The **Open** method on a **Recordset** object is used to open tables on a remote DDM server. Using the **Open** method on a **Recordset** object establishes the physical connection to a data source and opens a cursor that represents records from a base table or the results of a query. After this method successfully completes, the **Recordset** object is live and other methods can be invoked on the **Recordset** object to process results.

The optional *Source* parameter is used to specify the command text required to open a data file on the host using the OLE DB Provider for AS/400 and VSAM. The syntax in this case is as follows:

```
recordset = CreateObject("ADODB.Recordset.2.0")
recordset.Open "EXEC OPEN LIBRARY/FILE", connection, adOpenDynamic, adLockOptimistic, adCmdTe
xt
```

Using the OLE DB Provider for AS/400 and VSAM, the *Source* parameter represents a table name and uses one of the following host file naming conventions.

Host file type	File naming convention
VSAM Data Sets	DATASETNAME.FILENAME
Partitioned Data Sets	DATASETNAME.FILENAME(MEMBER)
OS/400 Files	LIBRARY/FILE
OS/400 Files	LIBRARY/FILE.NAME
OS/400 File Members	LIBRARY/FILE(MEMBER)
OS/400 File Members	LIBRARY/FILE.NAME(MEMBER)

Note that if a member of a library contains a dot in the member name, the member name must be surrounded by double quotes. For example, if the member name is NAMES.DAT, the proper syntax for command text used for the **Recordset.Open** method is:

```
recordset = CreateObject("ADODB.Recordset.2.0")
recordset.Open "EXEC OPEN LIBRARY/FILE(""NAMES.DAT"")", connection, adOpenDynamic, adLockOpti
mistic, adCmdText
```

Note the doubled quotes are required surrounding the member name in this example since the member name contains a period. The full path to the mainframe data set must be specified when using the OLE DB Provider for AS/400 and VSAM. In the example above, there are two path elements (LIBRARY/FILE) and one name element (NAMES.DAT).

Whenever a VSAM data set is allocated, it is given a unique name composed of one or more segments. Each segment of the data set name is joined by periods and represents a level of qualification. For example, the following data set has four segments that comprise the fully-qualified data set name (three path elements and one name element):

```
SAMPLES.DEMO.KSDS.TITLES
```

The high-level qualifier is SAMPLES. The low-level qualifier is TITLES. Each segment can be from 1-8 characters in length (the first character must be alphabetic, while the remainder can be alphanumeric or hyphens). The full data set name must be no more than 44 characters in length and contain no more than 22 segments.

The optional *ActiveConnection* parameter corresponds to the **ActiveConnection** property on a **Recordset** object and specifies on which connection to open the **Recordset** object. If a connection string definition is passed for this argument, ADO opens a new connection using the specified parameters. The value of this **ActiveConnection** property can be changed after opening the **Recordset** to send updates to another provider. The **ActiveConnection** property is set to **Nothing** (in Microsoft® Visual Basic®) to disconnect the recordset from the OLE DB Provider. If the optional *ActiveConnection* parameter is used to specify a connection string, this string must contain a series of *argument=value* statements separated by semicolons.

The **ActiveConnection** property on a **Recordset** object automatically inherits the value used for the *ActiveConnection* parameter. Therefore, the **ActiveConnection** property of the **Recordset** object can be set before opening the **Recordset** object, or the *ActiveConnection* parameter can be used to set or override the current connection parameters during the **Open** method call.

The *CursorType* parameter cannot be omitted using the OLE DB Provider for AS/400 and VSAM since this parameter defaults to **adOpenForwardOnly**, a *CursorType* that is not supported on the OLE DB Provider. The *CursorType* parameter must be set to **adOpenDynamic**, otherwise an error will occur and results will be unpredictable.

There are a number of different ways to open a recordset and connect to a data source. The **Open** method of the **Recordset** object can pass all of the appropriate connection information as part of the *ActiveConnection* parameter or by setting the **ActiveConnection** property of the **Recordset** object, if this information is known in advance. The syntax, in this case using the *ActiveConnection* parameter and the OLE DB Provider for AS/400 and VSAM, is as follows:

```
recordset = CreateObject("ADODB.Recordset.2.0")
recordset.Open "EXEC OPEN LIBRARY/FILE","Provider=SNAOLEDB;Data Source=REMLU;User ID=USERNAME
;Password=password;Local LU=LOCAL;Remote LU=DATABASE;ModeName=QPCSUPP;CCSID=37;CodePage=437",
adOpenDynamic, adLockOptimistic, adCmdText
```

> **Note** Not all of these parameters are required. The registry settings for the Data Source usually have default values set for remote LU, local LU, APPC mode, CCSID, and CodePage. If a data source is specified, this other information is not usually needed. These registry settings are configured by using the Microsoft Management Console snap-in for the OLE DB Provider for AS/400 and VSAM.

For the other parameters that correspond directly to the properties of a **Recordset** object (*Source*, *CursorType*, and *LockType*), the relationship of the parameters to the properties is as follows:

- The property is read/write before the **Recordset** object is opened.
- The property settings are used unless the corresponding parameters are passed when executing the **Open** method. If a parameter is passed, it overrides the corresponding property setting, and the property setting is updated with the parameter value.
- After the **Recordset** object is opened, these properties become read-only.

> **Note** For **Recordset** objects whose **Source** property is set to a valid **Command** object, the **ActiveConnection** property is read-only, even if the **Recordset** object is not open.

If a **Command** object is passed in the *Source* parameter and an *ActiveConnection* parameter is also passed, an error occurs. The **ActiveConnection** property of the **Command** object must already be set to a valid **Connection** object or connection string.

If a **Command** object is not passed in the *Source* argument, the *Options* argument must be set to **adCmdText**. If the *Options* argument is not defined, you may experience diminished performance because ADO must make calls to the OLE DB Provider to determine if the argument is a command statement. If you know what *Source* type you are using, setting the *Options* argument instructs ADO to jump directly to the relevant code.

If the data source returns no records, the provider sets both the **BOF** and **EOF** properties on the **Recordset** object to **True**, and the current record position is undefined. You can still add new data to this empty **Recordset** object if the cursor type allows it.

When operations have been concluded over an open **Recordset** object, the **Close** method should be invoked on the **Recordset** object to free any associated system resources. Closing a **Recordset** object does not remove it from memory; you may change its property settings and use the **Open** method to open it again later. To completely eliminate an object from memory, set the **Recordset** object variable to **Nothing**.

If errors occur, these can be examined with the **Errors** collection on the **Recordset** object.

OpenSchema Method

The **OpenSchema** method on a **Connection** object obtains database schema information from the provider.

```
recordset = connection.OpenSchema ( QueryType, Criteria, SchemaID )
```

Parameters

QueryType

This parameter specifies a **SchemaEnum** value that indicates the type of schema query to run.

The **SchemaEnum** values supported by the Microsoft® OLE DB Provider for AS/400 and VSAM are listed in a table following the Parameters section

Criteria

This optional parameter specifies an array of query constraints for each *QueryType* option, as listed below.

The values supported by the OLE DB Provider for AS/400 and VSAM can be one of the following constants depending on the *QueryType*:

> **Note** The **adSchemaindexes** TYPE restriction is not supported by the OLE DB Provider for DB2.
>
> The **adSchemaProcedures** PROCEDURE_SCHEMA, and **adSchemaProcedureParameters** PROCEDURE_SCHEMA restrictions are not supported when connecting to DB/2 on OS/390 platforms.

Values used for the Criteria parameter are listed in the table following the Parameters section.

SchemaID

This optional parameter specifies the GUID for a provider-schema schema query not defined by the OLE DB specification. This parameter is required if the *QueryType* parameter is set to **adSchemaProviderSpecific**; otherwise, it is not used. This parameter is not supported by the OLE DB Provider for AS/400 and VSAM.

Possible values used by *QueryType*

Enumeration	Value	Description
adSchemaColumns	4	This value indicates that the *QueryType* is requesting column information for tables on the server (not supported when connecting to mainframes).
adSchemaIndexes	12	This value indicates that the *QueryType* is requesting index information about the tables on the server (not supported when connecting to mainframes).
adSchemaTables	20	This value indicates that the *QueryType* is requesting information about the tables on the server.
adSchemaProviderTypes	22	This value indicates that the *QueryType* is requesting provider-type information.

The **SchemaEnum** values supported by the Microsoft® OLE DB Provider for DB2 and the Microsoft® ODBC Driver for DB2 can be one of the following constants:

Enumeration	Value	Description
adSchemaColumns	4	This value indicates that the *QueryType* is requesting column information for tables on the server (not supported when connecting to mainframes).
adSchemaIndexes	12	This value indicates that the *QueryType* is requesting index information about the tables on the server (not supported when connecting to mainframes).
adSchemaProcedures	16	This value indicates that the *QueryType* is requesting information about stored procedures on the server.
adSchemaTables	20	This value indicates that the *QueryType* is requesting information about the tables on the server.
adSchemaProviderTypes	22	This value indicates that the *QueryType* is requesting provider-type information.
adSchemaProcedureParameters	26	This value indicates that the *QueryType* is requesting information about parameters used by stored procedures on the server.
adSchemaPrimaryKeys	28	This value indicates that the *QueryType* is requesting information about the primary keys for tables on the server.

Values used by the Criteria parameter

QueryType / Enumeration
adSchemaColumns
TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
COLUMN_NAME
adSchemaIndexes
TABLE_CATALOG
TABLE_SCHEMA
INDEX_NAME
TYPE
TABLE_NAME
adSchemaTables
TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
TABLE_TYPE
adSchemaProviderTypes
DATA_TYPE
BEST_MATCH

The values supported by the OLE DB Provider for DB2 and the ODBC Driver for DB2 can be one of the following constants depending on the *QueryType*:

QueryType / Enumeration
adSchemaColumns
TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
COLUMN_NAME
adSchemaIndexes
TABLE_CATALOG
TABLE_SCHEMA
INDEX_NAME
TABLE_NAME
adSchemaPrimaryKeys
TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
adSchemaProcedures
PROCEDURE_CATALOG
PROCEDURE_SCHEMA (see Notes)
PROCEDURE_NAME
PROCEDURE_TYPE
adSchemaProcedureParameters
PROCEDURE_CATALOG
PROCEDURE_SCHEMA (see Notes)
PROCEDURE_NAME
PROCEDURE_TYPE
adSchemaProviderTypes
DATA_TYPE
BEST_MATCH
adSchemaTables

TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
TABLE_TYPE

Return Values

Returns a **Recordset** object that contains schema information requested.

Remarks

The **OpenSchema** method on a **Connection** object is used to return information about the data source, such as information about the tables on the server and the columns in the tables.

The *Criteria* argument is an array of values that can be used to limit the results of a schema query. Each schema query supports a different set of parameters. The actual schemas are defined by the OLE DB specification under the **IDBSchemaRowset** interface. The schema queries supported in ADO 1.5 and later by the OLE DB Provider for AS/400 and VSAM are listed above.

The **OpenSchema** method allows an application to pass at run time the target library of a Partioned Data Set (PDS/PDSE), a dataset, or a member name as one of the *Criteria* array arguments to retrieve the schema.

Providers are not required to support all of the OLE DB standard schema *QueryType* values. Specifically, only **adSchemaTables**, **adSchemaColumns**, and **adSchemaProviderTypes** are required by the OLE DB specification. However, the provider is not required to support the *Criteria* constraints listed above for those schema queries. Support for other schema *QueryType* values is optional.

The schema information specified in OLE DB is based on the assumption that providers support the concepts of a catalog and a schema. The ANSI SQL 92 specification defines them as follows:

- A catalog contains one or more schemas, but always contains a schema named INFORMATION_SCHEMA which contains the views and domains of the information schema. In Microsoft® SQL Server and Microsoft® Access terms, a catalog is a database; in ODBC 2.*x* terms, a catalog is a qualifier.
- A schema is a collection of database objects that are owned or have been created by a particular user. In Microsoft SQL Server and ODBC 2.*x* terms, a schema is an owner; there is no equivalent to a schema in a Microsoft Access database.

Schema information in ADO and OLE DB is retrieved using predefined schema rowsets. The following section lists the contents of each schema rowset supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

adSchemaColumns

The **adSchemaColumns** *QueryType* identifies the columns of tables defined in the catalog that are accessible to a given user. This *QueryType* is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaColumns** *QueryType* contains the following columns:

Column name	Type indicator	Description
TABLE_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Unqualified schema name. NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	Table name.
COLUMN_NAME	DBTYPE_WSTR	The name of the column; this might not be unique. If this cannot be determined, a NULL is returned. This column, together with the COLUMN_GUID and COLUMN_PROPID columns, forms the column ID. One or more of these columns will be NULL depending on which elements of the DBID structure the provider uses. If possible, the resulting column ID should be persistent. However, some providers do not support persistent identifiers for columns. The column ID of a base table should be invariant under views.
COLUMN_GUID	DBTYPE_GUID	Column GUID.
COLUMN_PROPID	DBTYPE_UI4	Column property ID.
ORDINAL_POSITION	DBTYPE_UI4	The ordinal of the column. Columns are numbered starting from one. NULL if there is no stable ordinal value for the column.
COLUMN_HASDEFAULT	DBTYPE_BOOL	VARIANT_TRUE: The column has a default value. VARIANT_FALSE: The column does not have a default value or it is unknown whether the column has a default value.

COLUMN_DEFAULT	DBTYPE_WSTR	Default value of the column. A provider may expose DBCOLUMN_DEFAULTVALUE but not DBCOLUMN_HASDEFAULT (for SQL 92 tables) in the rowset returned by **IColumnsRowset::GetColumnsRowset**. If the default value is the NULL value, COLUMN_HASDEFAULT is VARIANT_TRUE, and the COLUMN_DEFAULT column is a NULL value.
COLUMN_FLAGS	DBTYPE_UI4	A bitmask that describes column characteristics. The DBCOLUMNFLAGS enumerated type specifies the bits in the bit mask. For information about DBCOLUMNFLAGS, see **IColumnsInfo::GetColumnInfo**. This column cannot contain a NULL value.
IS_NULLABLE	DBTYPE_BOOL	VARIANT_TRUE: The column might be nullable. VARIANT_FALSE: The column is known not to be nullable.
DATA_TYPE	DBTYPE_UI2	The indicator of the column's data type. If the data type of the column varies from row to row, this must be DBTYPE_VARIANT.
TYPE_GUID	DBTYPE_GUID	The GUID of the column's data type.
CHARACTER_MAXIMUM_LENGTH	DBTYPE_UI4	The maximum possible length of a value in the column. For character, binary, or bit columns, this is one of the following: The maximum length of the column in characters, bytes, or bits, respectively, if one is defined. For example, a CHAR(5) column in an SQL table has a maximum length of five (5). The maximum length of the data type in characters, bytes, or bits, respectively, if the column does not have a defined length. Zero (0) if neither the column nor the data type has a defined maximum length. NULL for all other types of columns.
CHARACTER_OCTET_LENGTH	DBTYPE_UI4	Maximum length in octets (bytes) of the column, if the type of the column is character or binary. A value of zero means the column has no maximum length. NULL for all other types of columns.
NUMERIC_PRECISION	DBTYPE_UI2	If the column's data type is numeric, this is the maximum precision of the column. The precision of columns with a data type of DBTYPE_DECIMAL or DBTYPE_NUMERIC depends on the definition of the column. If the column's data type is not numeric, this is NULL.
NUMERIC_SCALE	DBTYPE_I2	If column's type indicator is DBTYPE_DECIMAL or DBTYPE_NUMERIC, this is the number of digits to the right of the decimal point. Otherwise, this is NULL.
DATETIME_PRECISION	DBTYPE_UI4	Datetime precision (number of digits in the fractional seconds portion) of the column if the column is a datetime or interval type.

CHARACTER_SET_CATALOG	DBTYPE_WSTR	Catalog name in which the character set is defined. NULL if the provider does not support catalogs or different character sets.
CHARACTER_SET_SCHEMA	DBTYPE_WSTR	Unqualified schema name in which the character set is defined. NULL if the provider does not support schemas or different character sets.
CHARACTER_SET_NAME	DBTYPE_WSTR	Character set name. NULL if the provider does not support different character sets.
COLLATION_CATALOG	DBTYPE_WSTR	Catalog name in which the collation is defined. NULL if the provider does not support catalogs or different collations.
COLLATION_SCHEMA	DBTYPE_WSTR	Unqualified schema name in which the collation is defined. NULL if the provider does not support schemas or different collations.
COLLATION_NAME	DBTYPE_WSTR	Collation name. NULL if the provider does not support different collations.
DOMAIN_CATALOG	DBTYPE_WSTR	Catalog name in which the domain is defined. NULL if the provider does not support catalogs or domains.
DOMAIN_SCHEMA	DBTYPE_WSTR	Unqualified schema name in which the domain is defined. NULL if the provider does not support schemas or domains.
DOMAIN_NAME	DBTYPE_WSTR	Domain name. NULL if the provider does not support domains.
DESCRIPTION	DBTYPE_WSTR	Human-readable description of the column. For example, the description for a column named Name in the Employee table might be "Employee name."

The default sort order for the **adSchemaColumns** rowset is TABLE_CATALOG, TABLE_SCHEMA, and TABLE_NAME.

adSchemaIndexes

The **adSchemaIndexes** *QueryType* identifies the indexes defined in the catalog that are owned by a given user. This *QueryType* is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaIndexes** *QueryType* contains the following columns:

Column name	Type indicator	Description
TABLE_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Unqualified schema name. NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	Table name.
INDEX_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
INDEX_SCHEMA	DBTYPE_WSTR	Unqualified schema name. NULL if the provider does not support schemas.
INDEX_NAME	DBTYPE_WSTR	Index name.
PRIMARY_KEY	DBTYPE_BOOL	Whether the index represents the primary key on the table. NULL if this is not known.
UNIQUE	DBTYPE_BOOL	Whether index keys must be unique. One of the following: VARIANT_TRUE: The index keys must be unique. VARIANT_FALSE: Duplicate keys are allowed.
CLUSTERED	DBTYPE_BOOL	Whether an index is clustered. One of the following: VARIANT_TRUE: The leaf nodes of the index contain full rows, not bookmarks. This is a way to represent a table clustered by key value. VARIANT_FALSE: The leaf nodes of the index contain bookmarks of the base table rows whose key value matches the key value of the index entry.
TYPE	DBTYPE_UI2	The type of the index. One of the following: DBPROPVAL_IT_BTREE: The index is a B+-tree. DBPROPVAL_IT_HASH: The index is a hash file using, for example, linear or extensible hashing. DBPROPVAL_IT_CONTENT: The index is a content index. DBPROPVAL_IT_OTHER: The index is some other type of index.
FILL_FACTOR	DBTYPE_I4	For a B+-tree index, this property represents the storage utilization factor of page nodes during the creation of the index. The value is an integer from 1 to 100 representing the percentage of use of an index node. For a linear hash index, this property represents the storage utilization of the entire hash structure (the ratio of used area to total allocated area) before a file structure expansion occurs.

INITIAL_SIZE	DBTYPE_I4	The total amount of bytes allocated to this structure at creation time.
NULLS	DBTYPE_I4	Whether null keys are allowed. One of the following: DBPROPVAL_IN_DISALLOWNULL: The index does not allow entries where the key columns are NULL. If the consumer attempts to insert an index entry with a NULL key, then the provider returns an error. DBPROPVAL_IN_IGNORENULL: The index does not insert entries containing NULL keys. If the consumer attempts to insert an index entry with a NULL key, then the provider ignores that entry and no error code is returned. DBPROPVAL_IN_IGNOREANYNULL: The index does not insert entries where some column key has a NULL value. For an index having a multicolumn search key, if the consumer inserts an index entry with NULL value in some column of the search key, then the provider ignores that entry and no error code is returned.
SORT_BOOKMARKS	DBTYPE_BOOL	How the index treats repeated keys. One of the following: VARIANT_TRUE: The index sorts repeated keys by bookmark. VARIANT_FALSE: The index does not sort repeated keys by bookmark.
AUTO_UPDATE	DBTYPE_BOOL	Whether the index is maintained automatically when changes are made to the corresponding base table. One of the following: VARIANT_TRUE: The index is automatically maintained. VARIANT_FALSE: The index must be maintained by the consumer through explicit calls to **IRowsetChange**. Ensuring consistency of the index as a result of updates to the associated base table is the responsibility of the consumer.
NULL_COLLATION	DBTYPE_I4	How NULLs are collated in the index. One of the following: DBPROPVAL_NC_END: NULLs are collated at the end of the list, regardless of the collation order. DBPROPVAL_NC_START: NULLs are collated at the start of the list, regardless of the collation order. DBPROPVAL_NC_HIGH: NULLs are collated at the high end of the list. DBPROPVAL_NC_LOW: NULLs are collated at the low end of the list.
ORDINAL_POSITION	DBTYPE_UI4	Ordinal position of the column in the index, starting with one.
COLUMN_NAME	DBTYPE_WSTR	Column name. This column, together with the COLUMN_GUID and COLUMN_PROPID columns, forms the column ID. One or more of these columns will be NULL depending on which elements of the DBID structure the provider uses.
COLUMN_GUID	DBTYPE_GUID	Column GUID.
COLUMN_PROPID	DBTYPE_UI4	Column property ID.
COLLATION	DBTYPE_I2	One of the following: DB_COLLATION_ASC: The sort sequence for the column is ascending. DB_COLLATION_DESC: The sort sequence for the column is descending. NULL: A column sort sequence is not supported.
CARDINALITY	DBTYPE_I4	Number of unique values in the index.
PAGES	DBTYPE_I4	Number of pages used to store the index.
FILTER_CONDITION	DBTYPE_WSTR	The WHERE clause identifying the filtering restriction.

The default sort order for the **adSchemaIndexes** rowset is UNIQUE, TYPE, INDEX_CATALOG, INDEX_SCHEMA, INDEX_NAME, and ORDINAL_POSITION.

adSchemaPrimaryKeys

The **adSchemaPrimaryKeys** *QueryType* identifies the primary key columns defined in the catalog by a given user. This *QueryType* is supported by the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaPrimaryKeys** *QueryType* contains the following columns:

TABLE_CATALOG	DBTYPE_WSTR	Catalog name in which the table is defined. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Unqualified schema name in which the table is defined. NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	Table name.
COLUMN_NAME	DBTYPE_WSTR	Primary key column name. This column, together with the COLUMN_GUID and COLUMN_PROPID columns, forms the column ID. One or more of these columns will be NULL depending on which elements of the DBID structure the provider uses.
COLUMN_GUID	DBTYPE_GUID	Primary key column GUID.
COLUMN_PROPID	DBTYPE_UI4	Primary key column property ID.
ORDINAL	DBTYPE_UI4	The order of the column names (and GUIDs and property IDs) in the key.
PK_NAME	DBTYPE_WSTR	Primary key name. NULL if the provider does not support primary key constraints.

The default sort order for the **adSchemaPrimaryKeys** rowset is UNIQUE, TABLE_CATALOG, TABLE_SCHEMA, and TABLE_NAME.

adSchemaProcedures

The **adSchemaProcedures** *QueryType* identifies information about the columns of rowsets returned by procedures. This *QueryType* is supported by the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaProcedures** *QueryType* contains the following columns:

Column name	Type indicator	Description
PROCEDURE_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
PROCEDURE_SCHEMA	DBTYPE_WSTR	Unqualified schema name. NULL if the provider does not support schemas.
PROCEDURE_NAME	DBTYPE_WSTR	Table name.
COLUMN_NAME	DBTYPE_WSTR	The name of the column; this might not be unique. If this cannot be determined, a NULL is returned. This column, together with the COLUMN_GUID and COLUMN_PROPID columns, forms the column ID. One or more of these columns will be NULL depending on which elements of the DBID structure the provider uses. If possible, the resulting column ID should be persistent. However, some providers do not support persistent identifiers for columns. The column ID of a base table should be invariant under views.
COLUMN_GUID	DBTYPE_GUID	Column GUID.
COLUMN_PROPID	DBTYPE_UI4	Column property ID.
ROWSET_NUMBER	DBTYPE_UI4	Number of the rowset containing the column. This is greater than one only if the procedure returns multiple rowsets.
ORDINAL_POSITION	DBTYPE_UI4	The ordinal of the column. Columns are numbered starting from one. NULL if there is no stable ordinal value for the column.

IS_NULLABLE	DBTYPE_BOOL	VARIANT_TRUE: The column might be nullable. VARIANT_FALSE: The column is known not to be nullable.
DATA_TYPE	DBTYPE_UI2	The indicator of the column's data type. If the data type of the column varies from row to row, this must be DBTYPE_VARIANT.
TYPE_GUID	DBTYPE_GUID	The GUID of the column's data type.
CHARACTER_MAXIMUM_LENGTH	DBTYPE_UI4	The maximum possible length of a value in the column. For character, binary, or bit columns, this is one of the following: The maximum length of the column in characters, bytes, or bits, respectively, if one is defined. For example, a CHAR(5) column in an SQL table has a maximum length of five (5). The maximum length of the data type in characters, bytes, or bits, respectively, if the column does not have a defined length. Zero (0) if neither the column nor the data type has a defined maximum length. NULL for all other types of columns.
CHARACTER_OCTET_LENGTH	DBTYPE_UI4	Maximum length in octets (bytes) of the column, if the type of the column is character or binary. A value of zero means the column has no maximum length. NULL for all other types of columns.
NUMERIC_PRECISION	DBTYPE_UI2	If the column's data type is numeric, this is the maximum precision of the column. The precision of columns with a data type of DBTYPE_DECIMAL or DBTYPE_NUMERIC depends on the definition of the column. If the column's data type is not numeric, this is NULL.
NUMERIC_SCALE	DBTYPE_I2	If column's type indicator is DBTYPE_DECIMAL or DBTYPE_NUMERIC, this is the number of digits to the right of the decimal point. Otherwise, this is NULL.
DESCRIPTION	DBTYPE_WSTR	Human-readable description of the column. For example, the description for a column named Name in the Employee table might be "Employee name."

The default sort order for the **adSchemaProcedures** rowset is PROCEDURE_CATALOG, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

adSchemaProcedureParameters

The **adSchemaProcedureParameters** *QueryType* identifies information about the parameters and return codes of procedures. This *QueryType* is supported by the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaProcedureParameters** *QueryType* contains the following columns:

Column name	Type indicator	Description
PROCEDURE_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
PROCEDURE_SCHEMA	DBTYPE_WSTR	Unqualified schema name. NULL if the provider does not support schemas.
PROCEDURE_NAME	DBTYPE_WSTR	Table name.
PARAMETER_NAME	DBTYPE_WSTR	Parameter name. NULL if the parameter is not named.
ORDINAL_POSITION	DBTYPE_UI2	If the parameter is an input, input/output, or output parameter, this is the one-based ordinal position of the parameter in the procedure call. If the parameter is the return value, this is zero.
PARAMETER_TYPE	DBTYPE_UI2	One of the following: DBPARAMTYPE_INPUT—The parameter is an input parameter. DBPARAMTYPE_INPUTOUTPUT—The parameter is an input/output parameter. DBPARAMTYPE_OUTPUT—The parameter is an output parameter. DBPARAMTYPE_RETURNVALUE—The parameter is a procedure return value. If the provider cannot determine the parameter type, this is NULL.
PARAMETER_HASDEFAULT	DBTYPE_BOOL	VARIANT_TRUE: The parameter has a default value. VARIANT_FALSE: The parameter does not have a default value or it is unknown whether the parameter has a default value.
PARAMETER_DEFAULT	DBTYPE_WSTR	Default value of the parameter. If the default value is the NULL value, PARAMETER_HASDEFAULT is VARIANT_TRUE, and the PARAMETER_DEFAULT value is a NULL value.
IS_NULLABLE	DBTYPE_BOOL	VARIANT_TRUE: The parameter might be nullable. VARIANT_FALSE: The parameter is not nullable.
DATA_TYPE	DBTYPE_UI2	The indicator of the parameter's data type.

CHARACTER_MAXIMUM_LENGTH	DBTYPE_UI4	The maximum possible length of a value in the parameter. For character, binary, or bit columns, this is one of the following: The maximum length of the parameter in characters, bytes, or bits, respectively, if one is defined. For example, a CHAR(5) column in an SQL table has a maximum length of five (5). The maximum length of the data type in characters, bytes, or bits, respectively, if the parameter does not have a defined length. Zero (0) if neither the parameter nor the data type has a defined maximum length. NULL for all other types of parameters.
CHARACTER_OCTET_LENGTH	DBTYPE_UI4	Maximum length in octets (bytes) of the parameter, if the type of the parameter is character or binary. A value of zero means the parameter has no maximum length. NULL for all other types of parameter.
NUMERIC_PRECISION	DBTYPE_UI2	If the parameter's data type is numeric, this is the maximum precision of the parameter. The precision of parameters with a data type of DBTYPE_DECIMAL or DBTYPE_NUMERIC depends on the definition of the parameters. If the parameter's data type is not numeric, this is NULL.
NUMERIC_SCALE	DBTYPE_I2	If parameter's type indicator is DBTYPE_DECIMAL or DBTYPE_NUMERIC, this is the number of digits to the right of the decimal point. Otherwise, this is NULL.
DESCRIPTION	DBTYPE_WSTR	Human-readable description of the parameter. For example, the description for a parameter named Name in a procedure that adds a new employee might be "Employee name."
TYPE_NAME	DBTYPE_WSTR	Provider-specific data type name.
LOCAL_TYPE_NAME	DBTYPE_WSTR	Localized version of TYPE_NAME. NULL is returned if a localized name is not supported by the data provider.

The default sort order for the **adSchemaProcedureParameters** rowset is PROCEDURE_CATALOG, PROCEDURE_SCHEMA, and PROCEDURE_NAME.

adSchemaProviderTypes

The **adSchemaProviderTypes** *QueryType* identifies the data types supported by the data provider. This *QueryType* is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaProviderType** *QueryType* contains the following columns:

Column name	Type indicator	Description
TYPE_NAME	DBTYPE_WSTR	Provider-specific data type name.
DATA_TYPE	DBTYPE_UI2	The indicator of the data type.
COLUMN_SIZE	DBTYPE_UI4	The length of a non-numeric column or parameter refers to either the maximum or the defined length for this type by the provider. For character data, this is the maximum or defined length in characters. If the data type is numeric, this is the upper bound on the maximum precision of the data type.
LITERAL_PREFIX	DBTYPE_WSTR	Character or characters used to prefix a literal of this type in a text command.

LITERAL_SUFFIX	DBTYPE_WSTR	Character or characters used to suffix a literal of this type in a text command.
CREATE_PARAMS	DBTYPE_WSTR	The creation parameters are specified by the consumer when creating a column of this data type. For example, the SQL data type DECIMAL needs a precision and a scale. In this case, the creation parameters might be the string "precision,scale". In a text command, to create a DECIMAL column with a precision of 10 and a scale of 2, the value of the TYPE_NAME column might be DECIMAL() and the complete type specification would be DECIMAL(10,2). The creation parameters appear as a comma-separated list of values, in the order they are to be supplied, with no surrounding parentheses. If a creation parameter is length, maximum length, precision, or scale, "length", "max length", "precision", and "scale" should be used, respectively. If the creation parameters are some other value, it is provider-specific what text is used to describe the creation parameter. If the data type requires creation parameters, "()" generally appears in the type name. This indicates the position at which to insert the creation parameters. If the type name does not include "()", the creation parameters are enclosed in parentheses and appended to the end of the data type name.
IS_NULLABLE	DBTYPE_BOOL	VARIANT_TRUE: The data type is nullable. VARIANT_FALSE: The data type is not nullable. NULL: It is not known whether the data type is nullable.
CASE_SENSITIVE	DBTYPE_BOOL	VARIANT_TRUE: The data type is a character type and is case-sensitive. VARIANT_FALSE: The data type is not a character type or is not case-sensitive.
SEARCHABLE	DBTYPE_UI4	If the provider supports **ICommandText**, then this column is an integer indicating the searchability of a data type, otherwise this column is NULL. One of the following: DB_UNSEARCHABLE: The data type cannot be used in a WHERE clause. DB_LIKE_ONLY: The data type can be used in a WHERE clause only with the LIKE predicate. DB_ALL_EXCEPT_LIKE: The data type can be used in a WHERE clause with all comparison operators except LIKE. DB_SEARCHABLE: The data type can be used in a WHERE clause with any comparison operator.

UNSIGNED_ATTRIBUTE	DBTYPE_BOOL	VARIANT_TRUE: The data type is unsigned. VARIANT_FALSE: The data type is signed. NULL: Not applicable to data type.
FIXED_PREC_SCALE	DBTYPE_BOOL	VARIANT_TRUE: The data type has a fixed precision and scale. VARIANT_FALSE: The data type does not have a fixed precision and scale.
AUTO_UNIQUE_VALUE	DBTYPE_BOOL	VARIANT_TRUE: Values of this type can be auto-incrementing. VARIANT_FALSE: Values of this type cannot be auto-incrementing.
LOCAL_TYPE_NAME	DBTYPE_WSTR	Localized version of TYPE_NAME. NULL is returned if a localized name is not supported by the data provider.

MINIMUM_SCALE	DBTYPE_I2	If the type indicator is DBTYPE_DECIMAL or DBTYPE_NUMERIC, this is the minimum number of digits allowed to the right of the decimal point. Otherwise, this is NULL.
MAXIMUM_SCALE	DBTYPE_I2	If the type indicator is DBTYPE_DECIMAL or DBTYPE_NUMERIC, this is the maximum number of digits allowed to the right of the decimal point. Otherwise, this is NULL.
GUID	DBTYPE_GUID	The GUID of the type. All types supported by a provider are described in a type library, so each type has a corresponding GUID.
TYPELIB	DBTYPE_WSTR	The type library containing the description of this type. All types supported by a provider are described in one or more type libraries.
VERSION	DBTYPE_WSTR	The version of the type definition. Providers may wish to version type definitions. Different providers may use different version schemes, such as a timestamp or number (integer or float). NULL if not supported.

IS_LONG	DBTYPE_BOOL	VARIANT_TRUE: The data type is a BLOB that contains very long data; the definition of very long data is provider-specific. VARIANT_FALSE: The data type is a BLOB that does not contain very long data or is not a BLOB. This value determines the setting of the DBCOLUMNFLAGS_ISLONG flag returned by **GetColumnInfo** in **IColumnsInfo** and **GetParameterInfo** in **ICommandWithParameters**. For more information, see **GetColumnInfo** and **GetParameterInfo**.
BEST_MATCH	DBTYPE_BOOL	VARIANT_TRUE: The data type is the best match between all data types in the data source and the OLE DB data type indicated by the value in the DATA_TYPE column. VARIANT_FALSE: The data type is not the best match. For each set of rows in which the value of the DATA_TYPE column is the same, the BEST_MATCH column is set to VARIANT_TRUE in only one row.

The default sort order for the **adSchemaProviderTypes** rowset is DATA_TYPE.

adSchemaTables

The **adSchemaTables** *QueryType* identifies the tables defined in the catalog that are accessible to a given user. This *QueryType* is supported by the Microsoft® OLE DB Provider for AS/400 and VSAM and the Microsoft® OLE DB Provider for DB2.

The rowset returned by an **adSchemaTables** *QueryType* contains the following columns:

Column name	Type indicator	Description
TABLE_CATALOG	DBTYPE_WSTR	Catalog name. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Unqualified Schema Name. NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	Table name.
TABLE_TYPE	DBTYPE_WSTR	Table type. One of the following or a provider-specific value. "ALIAS" "TABLE" "SYNONYM" "SYSTEM TABLE" "VIEW" "GLOBAL TEMPORARY" "LOCAL TEMPORARY"
TABLE_GUID	DBTYPE_GUID	GUID that uniquely identifies the table. Providers that do not use GUIDs to identify tables should return NULL in this column.
DESCRIPTION	DBTYPE_WSTR	Human-readable description of the table.

The default sort order for the **adSchemaTables** rowset is TABLE_TYPE, TABLE_CATALOG, TABLE_SCHEMA, and TABLE_NAME.

Precision Property

The **Precision** property on a **Field** object indicates the degree of precision for Numeric values for numeric **Field** objects. This property returns a byte value indicating the maximum number of digits used to represent numeric values in a **Field** object.

```
numericPrecision = currentfield.Precision
```

Remarks

The **Precision** property is used to return the precision of a numeric field object.

The byte value that the **Precision** property will return is dependent on the data type of the **Field** object. The value for the ADO data type of the **Field** object can be one of the following enumerated values for **DataTypeEnum**:

Enumeration	Value	Description
adEmpty	0	This data type indicates that no value was specified (DBTYPE_EMPTY).
adSmallInt	2	This data type indicates a two-byte (16-bit) signed integer (DBTYPE_I2).
adInteger	3	This data type indicates a four-byte (32-bit) signed integer (DBTYPE_I4).
adSingle	4	This data type indicates a four-byte (32-bit) single precision IEEE floating point number (DBTYPE_R4).
adDouble	5	This data type indicates an eight-byte (64-bit) double precision IEEE floating point number (DBTYPE_R8).
adCurrency	6	A data type indicates a currency value (DBTYPE_CY). Currency is a fixed-point number with 4 digits to the right of the decimal point. It is stored in an eight-byte signed integer scaled by 10,000. This data type is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
adDate	7	This data type indicates a date value stored as a Double, the whole part of which is the number of days since December 30, 1899, and the fractional part of which is the fraction of a day. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBSTR	8	This data type indicate a null-terminated Unicode character string (DBTYPE_BSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIDispatch	9	This data type indicates a pointer to an IDispatch interface on an OLE object (DBTYPE_IDISPATCH). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adError	10	This data type indicates a 32-bit error code (DBTYPE_ERROR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBoolean	11	This data type indicates a Boolean value (DBTYPE_BOOL). This data type is not supported by the OLE DB Provider for AS/400 and VSAM.
adVariant	12	This data type indicates an automation variant (DBTYPE_VARIANT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIUnknown	13	This data type indicates a pointer to an IUnknown interface on an OLE object (DBTYPE_IUNKNOWN). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDecimal	14	This data type indicates numeric data with a fixed precision and scale (DBTYPE_DECIMAL).
adTinyInt	16	This data type indicates a single -byte (8-bit) signed integer (DBTYPE_I1). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adUnsignedTinyInt	17	This data type indicates a single-byte (8-bit) unsigned integer (DBTYPE_UI1). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adUnsignedSmallInt	18	This data type indicates a two-byte (16-bit) unsigned integer (DBTYPE_UI2). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

adUnsignedInt	19	This data type indicates a four-byte (32-bit) unsigned integer (DBTYPE_UI4). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBigInt	20	This data type indicates an eight-byte (64-bit) signed integer (DBTYPE_I8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM.
adUnsignedBigInt	21	This data type indicates an eight-byte (64-bit) unsigned integer (DBTYPE_UI8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adGUID	72	This data type indicates a globally unique identifier or GUID (DBTYPE_GUID). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBinary	128	This data type indicates fixed-length binary data (DBTYPE_BYTES).
adChar	129	This data type indicates a character string value (DBTYPE_STR).
adWChar	130	This data type indicates a null-terminated Unicode character string (DBTYPE_WSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adNumeric	131	This data type indicates numeric data where the precision and scale are exactly as specified (DBTYPE_NUMERIC).
adUserDefined	132	This data type indicates user-defined data (DBTYPE_UDT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDBDate	133	This data type indicates a OLE DB date structure (DBTYPE_DATE).
adDBTime	134	This data type indicates a OLE DB time structure (DBTYPE_TIME).
adDBTimeStamp	135	This data type indicates a OLE DB timestamp structure (DBTYPE_TIMESTAMP).
adVarChar	200	This data type indicates variable-length character data (DBTYPE_STR).
adLongVarChar	201	This data type indicates a long string value.
adVarWChar	202	This data type indicates a Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adLongVarWChar	203	This data type indicates a long Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adVarBinary	204	This data type indicates variable-length binary data (DBTYPE_BYTES).
adLongVarBinary	205	This data type indicates a long binary value.

Note that the **Precision** property returns values that differ from the precision of the host data type for the following ADO data types:

ADO Data Type	Comments
adSmallInt	The precision on the host is 4, but the OLE DB Provider returns a precision of 5.
adInteger	The precision on the host is 8, but the OLE DB Provider returns a precision of 10.

adSingle	The precision on the host is 9, but the OLE DB Provider returns a precision of 7.
adDouble	The precision on the host is 17, but the OLE DB Provider returns a precision of 15.

OriginalValue Property

The **OriginalValue** property on a **Field** object indicates the value of a **Field** that existed in the record before any changes were made. This property returns a Variant.

```
oldValue = currentfield.OriginalValue
```

Remarks

The **OriginalValue** property is used to return the original field value for a field from the current record.

In immediate update mode (the provider writes changes to the underlying data source once the **Update** method is called), the **OriginalValue** property returns the field value that existed prior to any changes (that is, since the last **Update** method call). This is the same value that the **CancelUpdate** method uses to replace the **Value** property.

In batch update mode (the provider caches multiple changes and writes them to the underlying data source only when the **UpdateBatch** method is called), the **OriginalValue** property returns the field value that existed prior to any changes (that is, since the last **UpdateBatch** method call). This is the same value that the **CancelBatch** method uses to replace the **Value** property. When this property is used with the **UnderlyingValue** property, you can resolve conflicts that arise from batch updates.

Provider Property

The **Provider** property on a **Connection** object indicates the name of the provider. This property sets or returns a String value.

```
oldProvider = currentConnection.Provider
currentConnection.Provider = "SNAOLEDB"
```

Remarks

The **Provider** property is used to set or return the name of the provider for the connection. This property can also be set by the contents of **ConnectionString** property or the **ConnectionString** argument of the **Open** method. However, specifying a provider in more than one place while calling the **Open** method can have unpredictable results. If no provider is specified, the property will default to MSDASQL (Microsoft® OLE DB Provider for ODBC).

The Microsoft® OLE DB Provider for AS/400 and VSAM requires "SNAOLEDB" as the **Provider** property string.

The Microsoft® OLE DB Provider for DB2 requires "DB2OLEDB" as the **Provider** property string.

The **Provider** property is read/write when the connection is closed and read-only when it is open. The setting does not take effect until either the **Connection** object is opened or the **Properties** collection of the **Connection** object is accessed. If the setting is invalid, an error occurs.

Refresh Method

The **Refresh** method on a **Collection** object updates the objects in a collection to reflect objects available from and specific to the OLE DB provider.

```
collection.Refresh
```

Parameters

None.

Remarks

This method is only supported on the **Field**s and **Properties** collections under the Microsoft® OLE DB Provider for AS/400 and VSAM.

The **Refresh** method accomplishes different tasks depending on the collection object on which it is called.

Using the **Refresh** method on the **Field**s collection has no visible effect. To retrieve changes from the underlying database structure, either the **Requery** method must be used or, if the **Recordset** object does not support bookmarks, the **MoveFirst** method must be used.

Using the **Refresh** method on a **Properties** collection of some objects populates the collection with the dynamic properties the provider exposes. These properties provide information about features specific to the provider beyond the built-in properties ADO supports. The OLE DB Data Provider for AS/400 and VSAM does not support any provider-specific properties.

Requery Method

The **Requery** method on a **Recordset** object updates the data in a **Recordset** object by re-executing the query on which the object is based.

```
recordset.Requery
```

Parameters

None.

Remarks

The **Requery** method is used to refresh the entire contents of a **Recordset** object from the data source by reissuing the original command and retrieving the data a second time. Calling this method is equivalent to calling the **Close** and **Open** methods in succession. If you are editing the current record or adding a new record, an error occurs.

While the **Recordset** object is open, the properties that define the nature of the cursor (**CursorType**, **LockType**, **MaxRecords**, and other properties) are read-only. Thus, the **Requery** method can only refresh the current cursor. To change any of the cursor properties and view the results, the **Close** method must be used so that the properties become read/write again. You can then change the property settings and call the **Open** method to reopen the cursor.

Save Method

The **Save** method on a **Recordset** object saves the Recordset in a file or Stream object.

```
recordset.Save Destination, Persistent Format
```

Parameters

Destination
This optional parameter specifies a Variant representing the complete path name of the file where the **Recordset** is to be saved, or a reference to a **Stream** object.

Persistent Format
This optional parameter specifies a Long integer value representing a **PersistFormatEnum** value that specifies the format in which the **Recordset** is to be saved (XML or ADTG). The default value is **adPersistADTG**.

The **PersistFormatEnum** value can be one of the following constants:

Enumeration	Value	Description
adPersistADTG	0	This value indicates Microsoft Advanced Data TableGram (ADTG) format.
adPersistXML	1	This value indicates Extensible Markup Language (XML) format.

Remarks

The **Save** method can only be invoked on an open **Recordset**. Use the **Open** method to later restore the **Recordset** from *Destination*.

When using the Microsoft® OLE DB Provider for AS/400 and VSAM and the Filter property is in effect for the **Recordset**, then only the rows accessible under the filter are saved.

The first time you save the **Recordset**, it is optional to specify *Destination*. If the *Destination* parameter is omitted, a new file will be created with a name set to the value of the Source property of the **Recordset**.

The *Destination* parameter should be omitted when you subsequently call **Save** after the first save, or a run-time error will occur. If you subsequently call **Save** with a new *Destination*, the **Recordset** is saved to the new destination. However, the new destination and the original destination will both be open.

Save does not close the **Recordset** or *Destination*, so you can continue to work with the **Recordset** and save your most recent changes. *Destination* remains open until the **Recordset** is closed, during which time other applications can read but not write to *Destination*.

For reasons of security, the **Save** method permits only the use of low and custom security settings from a script executed by Microsoft® Internet Explorer. For a more detailed explanation of security issues, see "ADO and RDS Security Issues in Microsoft Internet Explorer" found in the ActiveX® Data Objects (ADO) Technical Articles of the Microsoft Data Access Technical Articles.

If the **Save** method is called while an asynchronous **Recordset** fetch, execute, or update operation is in progress, then **Save** waits until the asynchronous operation is complete.

Records are saved beginning with the first row of the **Recordset**. When the **Save** method is finished, the current row position is moved to the first row of the **Recordset**.

For best results, set the CursorLocation property to **adUseClient** with **Save**. If your provider does not support all of the features necessary to save **Recordset** objects, the Cursor Service will provide these features.

When a **Recordset** is persisted with the **CursorLocation** property set to **adUseServer**, the update capability for the **Recordset** is limited. Typically, only single-table updates, insertions, and deletions are allowed (dependent on features supported by the provider). The **Resync** method is also unavailable in this configuration.

Note that saving a **Recordset** with **Fields** of type **adVariant**, **adIDispatch**, or **adIUnknown** is not supported by ADO and can cause unpredictable results.

Because the *Destination* parameter can accept any object that supports the OLE DB IStream interface, a **Recordset** can be saved directly to the ASP Response object. For more information, see the XML Recordset Persistence Scenario in the *ADO Programmer's Reference*.

Sort Property

The **Sort** property on a **Recordset** object indicates that a recordset should be sorted.

```
Recordset.Sort BSTR Criteria
```

Parameters

Criteria

This parameter specifies the criteria used for sorting the **Recordset** object. This **Sort** property contains a comma-delimited list of column names and a direction specifier (ascending or descending) to be used for sorting records in a **Recordset** object. The direction specifier is a string (ASC or DESC). When a direction is not specified, the direction defaults to ascending.

An example of a **Sort** property criteria is as follows:

"LastName ASC, FirstName DESC, Initial"

Remarks

The **Sort** property is not supported by the OLE DB Provider for DB2 or the ODBC Driver for DB2.

The **Sort** property is used with an open **Recordset** object based on an AS/400 physical file. The **Sort** property allows the user to indicate which logical view to apply to an AS/400 physical file. The logical view must be a valid index specified in the description of the AS/400 physical file. The logical view is provided by the AS/400 logical file. The Microsoft® OLE DB Provider for AS/400 and VSAM responds to a **Sort** request by first closing the open physical file, and then opening the logical file that points back to the data in the physical file.

The **Recordset Sort** property is only supported on AS/400 hosts. If the user opens a **Recordset** object based on an AS/400 logical file, then there is likely no need to use **Recordset**.**Sort**. For performance reasons, applications should be written to open the AS/400 logical file first, because the overhead is so much greater when opening a physical file first.

If the CursorLocation property is set to **adUseClient** (use the client cursor engine), the **Sort** property will work if MDAC 2.0 or later is installed but will not work properly with earlier versions of ADO.

Source Property on Error Object

The **Source** property on a **Error** object indicates the name of the object or application that originally generated an error. This property returns a String value that indicates the name of an object or application.

```
errorSource = currentError.Source
```

Remarks

The **Source** property on a **Error** object is used to determine the name of the object or application that originally generated an error. This could be the object's class name or programmatic ID.

For errors in ADO, the property value will be **ADODB.***ObjectName*, where *ObjectName* is the name of the object that triggered the error. For ADOX and ADO MD, the value will be **ADOX.***ObjectName* and **ADOMD.***ObjectName,* respectively.

Based on the error documentation from the **Source**, Number, and Description properties of **Error** objects, you can write code that will handle the error appropriately.

The **Source** property is read-only for **Error** objects.

Source Property on Recordset Object

The **Source** property on a **Recordset** object indicates the data source for the Recordset object. This property sets a String value or **Command** object reference or returns only a String value that indicates the source of the **Recordset**.

```
currentSource = currentRecordset.Source
recordset.Source = newSource
```

Remarks

The **Source** property on a Recordset is used to specify a data source for a **Recordset** object.

Using the Microsoft® OLE DB Provider for AS/400 and VSAM, the **Source** property can be either a **Command** object variable or a table name.

Using the Microsoft® OLE DB Provider for DB2, the Source property can be one of the following: a **Command** object variable, an SQL statement, or a stored procedure. If the **Source** property is an SQL statement or a stored procedure, you can optimize performance by passing the appropriate *Options* argument with the **Open** method call.

If you set the **Source** property to a **Command** object, the **ActiveConnection** property of the **Recordset** object will inherit the value of the **ActiveConnection** property for the specified **Command** object. However, reading the **Source** property does not return a **Command** object; instead, it returns the **CommandText** property of the **Command** object to which you set the **Source** property.

The **Source** property is read/write for closed **Recordset** objects and read-only for open **Recordset** objects.

State Property

The **State** property on a **Connection**, **Command**, or **Recordset** object describes the current state of an object. This property sets or returns a Long value.

```
oldState = currentConnection.State
currentConnection.State = adStateClosed
```

Remarks

The **State** property is used to set or return the current state of an object. The value of the **State** property can be one of the following enumerated values:

Enumeration	Value	Description
adStateClosed	0	This value indicates that the object is closed. This is the default value.
AdStateOpen	1	This value indicates that the object is open.

The **State** property can be used to determine the current state of a given object at any time.

Status Property

The **Status** property on a **Recordset** object indicates the status of the current record with respect to batch updates or other bulk operations. This property returns a Long value.

```
oldStatus = currentRecordset.Status
```

Remarks

The **Status** property is used to return the current status of a recordset object at any time. The value of the **Status** property returns a sum of the following RecordStatusEnum enumerated values:

Enumeration	Value	Description
adRecOK	0	This value indicates that the recordset object was successfully updated.
AdRecNew	0x1	This value indicates that the recordset object is new.
AdRecModified	0x2	This value indicates that the recordset object was modified.
AdRecDeleted	0x4	This value indicates that the recordset object was deleted.
AdRecUnmodified	0x8	This value indicates that the recordset object was not modified.
adRecInvalid	0x10	This value indicates that the recordset object was not saved because its bookmark is invalid.
adRecMultipleChanges	0x40	This value indicates that the recordset object was not saved because it would have affected multiple records.
adRecPendingChanges	0x80	This value indicates that the recordset object was not saved because it refers to a pending insert.
adRecCanceled	0x100	This value indicates that the recordset object was not saved because the operation was canceled.
adRecCantRelease	0x400	This value indicates that the new recordset object was not saved because of existing record locks.
adRecConcurencyViolation	0x800	This value indicates that the recordset object was not saved because optimistic concurrency was in use.
adRecIntegrityViolation	0x1000	This value indicates that the recordset object was not saved because the user violated integrity constraints.
adRecMaxChangesExceeded	0x2000	This value indicates that the recordset object was not saved because there were too many pending changes.
adRecObjectOpen	0x4000	This value indicates that the recordset object was not saved because of a conflict with an open storage object.
adRecOutOfMemory	0x8000	This value indicates that the recordset object was not saved because the computer has run out of memory.
adRecPermissionDenied	0x10000	This value indicates that the recordset object was not saved because the user has insufficient permissions.
adRecSchemaViolation	0x20000	This value indicates that the recordset object was not saved because it violates the structure of the underlying database.
adRecDBDeleted	0x40000	This value indicates that the recordset object has already been deleted from the data source.

Use the **Status** property to see what changes are pending for records modified during batch updating. You can also use the **Status** property to view the status of records that fail during bulk operations such as when you call the **Resync**, **UpdateBatch**, or **CancelBatch** methods on a recordset object, or set the **Filter** property on a recordset object to an array of bookmarks. With this property, you can determine how a given record failed and resolve it accordingly.

Supports Method

The **Supports** method on a **Recordset** object determines whether a specified **Recordset** object supports a particular type of feature.

```
boolean = recordset.Supports ( CursorOptions )
```

Parameters

CursorOptions
This parameter specifies a Long expression that consists of one or more of the **CursorOptionEnum** values indicating which feature is being queried. The **CursorOptionEnum** value can be one of the constants shown in the table following the Parameters section.

Values for the CursorOptions parameter

Enumeration	Value	Description
adAddNew	0x1000400	This value indicates whether the **AddNew** method can be used to add new records.
adApproxPosition	0x4000	This value indicates whether the **AbsolutePosition** and **AbsolutePage** properties can read and set.
adBookMark	0x2000	This value indicates whether the **Bookmark** property can be used to access specific records.
adDelete	0x1000800	This value indicates whether the **Delete** method can be used to delete records.
adFind	0x80000	This value indicates whether the **Find** method can be used to locate a row in a **Recordset**.
adHoldRecords	0x100	This value indicates whether you can retrieve more records or change the next retrieve position without committing all pending changes.
adIndex	0x100000	This value indicates whether the **Index** property can be name an index.
adMovePrevious	0x200	This value indicates whether the **MoveFirst** and **MovePrevious** methods, and **Move** or **GetRows** methods can be used to move the current record position backward without requiring bookmarks.
adNotify	0x40000	This value indicates that the underlying data provider supports notifications (which determines whether **Recordset** events are supported).
adResync	0x20000	This value indicates whether the recordset cursor can be updated with the data visible in the underlying database using the **Resync** method.
adSeek	0x200000	This value indicates whether the **Seek** method can be used to locate a row in a **Recordset**.
adUpdate	0x18000	This value indicates whether the **Update** method can be used to modify existing data.
adUpdateBatch	0x10000	This value indicates whether batch updating can be used on the recordset (the **UpdateBatch** and **CancelBatch** methods) to transmit changes to the provider in groups.

Return Values

Returns a Boolean value that indicates whether all of the features identified by the *CursorOptions* argument are supported by the provider.

Remarks

The **Supports** method is used to determine what types of features (methods and properties) a **Recordset** object supports. If the **Recordset** object supports the features whose corresponding constants are in *CursorOptions*, the **Supports** method returns **True**. Otherwise, it returns **False**.

Although the **Supports** method may return **True** for a given feature, it does not guarantee that the OLE DB Provider can make the feature available under all circumstances. The **Supports** method simply returns whether the provider can support the specified function assuming certain conditions are met. For example, the **Supports** method may indicate that a **Recordset** object supports updates even though the cursor is based on a multitable join, some columns of which are not updatable.

Type Property

The **Type** property on a **Field** object indicates the operational type or data type for **Field** or **Property** objects. This property sets or returns a **DataTypeEnum** value.

```
datatype = currentfield.Type
```

Remarks

The **Type** property is used to return the data type of a numeric field object.

The value returned by the **Type** property on a **Field** object can be one of the following enumerated values for **DataTypeEnum**:

Enumeration	Value	Description
adEmpty	0	This data type indicates that no value was specified (DBTYPE_EMPTY).
AdSmallInt	2	This data type indicates a two-byte (16-bit) signed integer (DBTYPE_I2).
AdInteger	3	This data type indicates a four-byte (32bit) signed integer (DBTYPE_I4).
AdSingle	4	This data type indicates a four-byte (32-bit) single-precision IEEE floating-point number (DBTYPE_R4).
AdDouble	5	This data type indicates an eight-byte (64-bit) double-precision IEEE floating-point number (DBTYPE_R8).
adCurrency	6	A data type indicates a currency value (DBTYPE_CY). Currency is a fixed-point number with four digits to the right of the decimal point. It is stored in an eight-byte signed integer scaled by 10,000. This data type is not supported by the Microsoft® OLE DB Provider for AS/400 and VSAM or the Microsoft® OLE DB Provider for DB2.
adDate	7	This data type indicates a date value stored as a Double, the whole part of which is the number of days since December 30, 1899, and the fractional part of which is the fraction of a day. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBSTR	8	This data type indicates a null-terminated Unicode character string (DBTYPE_BSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIDispatch	9	This data type indicates a pointer to an **IDispatch** interface on an OLE object (DBTYPE_IDISPATCH). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adError	10	This data type indicates a 32-bit error code (DBTYPE_ERROR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBoolean	11	This data type indicates a Boolean value (DBTYPE_BOOL). This data type is not supported by the OLE DB Provider for AS/400 and VSAM.
adVariant	12	This data type indicates an Automation variant (DBTYPE_VARIANT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adIUnknown	13	This data type indicates a pointer to an **IUnknown** interface on an OLE object (DBTYPE_IUNKNOWN). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDecimal	14	This data type indicates numeric data with a fixed precision and scale (DBTYPE_DECIMAL).
adTinyInt	16	This data type indicates a single-byte (8-bit) signed integer (DBTYPE_I1). This data type is not supported by the OLE DB Provider.
adUnsignedTinyInt	17	This data type indicates a singlebyte (8-bit) unsigned integer (DBTYPE_UI1). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adUnsignedSmallInt	18	This data type indicates a two-byte (16-bit) unsigned integer (DBTYPE_UI2). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.

adUnsignedInt	19	This data type indicates a four-byte (32-bit) unsigned integer (DBTYPE_UI4). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBigInt	20	This data type indicates an eight-byte (64-bit) signed integer (DBTYPE_I8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM.
adUnsignedBigInt	21	This data type indicates an eight-byte (64-bit) unsigned integer (DBTYPE_UI8). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adGUID	72	This data type indicates a globally unique identifier or GUID (DBTYPE_GUID). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adBinary	128	This data type indicates fixed-length binary data (DBTYPE_BYTES).
adChar	129	This data type indicates a character string value (DBTYPE_STR).
adWChar	130	This data type indicates a null-terminated Unicode character string (DBTYPE_WSTR). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adNumeric	131	This data type indicates numeric data where the precision and scale are exactly as specified (DBTYPE_NUMERIC).
adUserDefined	132	This data type indicates user-defined data (DBTYPE_UDT). This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adDBDate	133	This data type indicates a OLE DB date structure (DBTYPE_DATE).
adDBTime	134	This data type indicates a OLE DB time structure (DBTYPE_TIME).
adDBTimeStamp	135	This data type indicates a OLE DB timestamp structure (DBTYPE_TIMESTAMP).
adVarChar	200	This data type indicates variable-length character data (DBTYPE_STR).
adLongVarChar	201	This data type indicates a long string value.
adVarWChar	202	This data type indicates a Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adLongVarWChar	203	This data type indicates a long Unicode string value. This data type is not supported by the OLE DB Provider for AS/400 and VSAM or the OLE DB Provider for DB2.
adVarBinary	204	This data type indicates variable-length binary data (DBTYPE_BYTES).
adLongVarBinary	205	This data type indicates a long binary value.

The corresponding OLE DB type indicator is shown in parentheses in the description column of the above table. For more information on OLE DB data types, see the *OLE DB 2.0 Programmer's Reference*.

UnderlyingValue Property

The **UnderlyingValue** property on a **Field** object indicates the **Field** object's current value in the database. This property returns a Variant.

```
actualValue = currentfield.UnderlyingValue
```

Remarks

The **UnderlyingValue** property is used to return the current field value from the database. The field value in the **UnderlyingValue** property is the value that is visible to your transaction and may be the result of a recent update by another transaction. This may differ from the **OriginalValue** property, which reflects the value that was originally returned to the **Recordset**.

This is similar to the affect of calling the **Resync** method, however the **UnderlyingValue** property returns only the value for a specific field from the current record. This is the same value that the **Resync** method uses to replace the **Value** property.

When this property is used with the **OriginalValue** property, you can resolve conflicts that arise from batch updates.

Update Method

The **Update** method on a **Recordset** object saves any changes you make to the current record of a **Recordset** object.

```
recordset.Update Fields, Values
```

Parameters

Fields
This optional parameter specifies a Variant representing a single name or a Variant array representing names or ordinal positions of the field or fields you wish to modify.

Values
This optional parameter specifies a Variant representing a single value or a Variant array representing values for the field or fields in the new record.

Remarks

The **Update** method is used to save any changes you make to the current record of a **Recordset** object since calling the **AddNew** method or since changing any field values in an existing record. The **Recordset** object must support updates for the **Update** method to be used successfully.

To set field values, do one of the following:

- Assign values to a **Field** object's **Value** property and call the **Update** method.
- Pass a field name and a value as arguments with the **Update** call.
- Pass an array of field names and an array of values with the **Update** call.

When arrays of fields and values are used, there must be an equal number of elements in both arrays. Also, the order of field names must match the order of field values. If the number and order of fields and values do not match, an error occurs.

If the **Recordset** object supports batch updating, then multiple changes to one or more records can be cached locally when the **UpdateBatch** method is called. If you are editing the current record or adding a new record when the **UpdateBatch** method is called, ADO will automatically call the **Update** method to save any pending changes to the current record before transmitting the batched changes to the OLE DB Provider.

If you move from the record you are adding or editing before calling the **Update** method, ADO will automatically call **Update** to save the changes. The **CancelUpdate** method must be called if you want to cancel any changes made to the current record or to discard a newly added record.

The current record remains current after the **Update** method is called.

UpdateBatch Method

The **UpdateBatch** method on a **Recordset** object writes all pending batch updates to the host.

```
recordset.UpdateBatch AffectedRecords
```

Parameters

AffectedRecords

This optional parameter specifies an **AffectEnum** value that determines how many records the **UpdateBatch** method will affect.

The **AffectEnum** value can be one of the following constants:

Enumeration	Value	Description
adAffectCurrent	1	This value writes pending changes only for the current record.
adAffectGroup	2	This value writes pending changes for the records that satisfy the current Filter property setting. You must set the **Filter** property to one of the valid predefined constants to use this option.
adAffectAll	3	This value writes pending changes for all the records in the **Recordset** object, including any hidden by the current **Filter** property setting. This value is the default.

Remarks

The **UpdateBatch** method is used when modifying a **Recordset** object in batch update mode to transmit all changes made in a **Recordset** object to the underlying database.

If the **Recordset** object supports batch updating, then multiple changes to one or more records can be cached locally until the **UpdateBatch** method is called. If you are editing the current record or adding a new record when the **UpdateBatch** method is called, ADO will automatically call the **Update** method to save any pending changes to the current record before transmitting the batched changes to the provider.

If the attempt to transmit changes fails because of a conflict with the underlying data (for example, a record has already been deleted by another user), the provider returns warnings to the **Errors** collection but does not halt program execution. A run-time error occurs only if there are conflicts on all the requested records. Use the **Filter** property (**adFilterAffectedRecords**) and the **Status** property to locate records with conflicts.

To cancel all pending batch updates, use the **CancelBatch** method.

Value Property

The **Value** property on a **Field** object indicates the value assigned to a **Field** or **Property** object. This property sets or returns a Variant value. The default value depends on the **Type** property of the **Field** object.

```
oldValue = currentfield.Value
currentField.Value = newValue
```

Remarks

The **Value** property is used to set or return data from **Field** objects or to set or return property settings with **Property** objects. Whether the **Value** property is read/write or read-only depends upon numerous factors. For a **Field** object, this includes whether the **Recordset** was opened as read-only or read/write.

ADO allows setting and returning long binary data with the **Value** property.from the database.

Version Property

The **Version** property on a **Connection** object indicates the ADO version number. This property returns a String value.

```
versionADO = currentConnection.Version
```

Remarks

The **Version** property is used to return the version number of the ADO implementation. The version of the provider will be available as a dynamic property in the **Properties** collection.

Data Queue ActiveX Control Reference

This section provides reference information on specific ActiveX® methods, properties, and event notifications supported by the Microsoft® Data Queue ActiveX Control. The function syntax and code examples are based on Microsoft Visual Basic.

AddQueueItem Method

The **AddQueueItem** method on an **IEGDataQueue** object adds a record to a specified data queue using the Data Queue ActiveX Control.

```
object.AddQueueItem QueueItem, fBlock
```

Parameters

QueueItem
This required parameter representing an **IEIGDataQueueItem** object instance specifying the queue item to add to the queue.
fBlock
This optional parameter specifies whether the operation should block until the completion status is known. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the fBlock Values table which follows the Parameters section.

***fBlock* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the operation should block until the completion status is known.
eigAnswerNo	1	This value indicates that the operation should not block. This is the default value for this parameter if it is omitted from the method call.

Remarks

The queue the item is sent to is represented by **QItemType** property on the **IEIGDataQueueItem**. The *QueueItem* is an object of type IEIGDataQueueItem that may have been initialized by the user or returned by a call to **GetQueueItem**.

Cancel Method

The **Cancel** method on an **IEGDataQueue** object terminates a transfer operation that is already in progress. This method cancels an item transfer using the Data Queue ActiveX Control.

```
object.Cancel
```

Parameters

None.

Remarks

The **Cancel** method is used to cancel a transfer operation that is already in progress.

CancelQueue Method

The **CancelQueue** method on an **IEGDataQueue** object indicates that an application using the Data Queue ActiveX Control no longer wants to be notified of an incoming queue data item. This can be used to stop pending notifications that were queued as a result of calling **GetQueueItem**.

```
object.CancelQueue CancelReqCount, CancelReqType, KeyValue
```

Parameters

CancelReqCount
This optional parameter is a count (a short value) of the number of outstanding queue receive requests to cancel. This parameter defaults to a value of 1.

CancelReqType
This optional parameter specifies the type of queue request to cancel. This parameter can be set to one of the **eigQItemTypeEnum** constants shown in the CancelReqType Values table which follows the Parameters section.

KeyValue
This optional parameter is a VARIANT representing the key value and has no default value. If the *CancelReqType* is eigKQItem, then this value specifies the key value to stop waiting on.

***CancelReqType* Values**

Enumeration	Value	Description
eigKQItem	0	A keyed item.
eigQItem	1	A non-keyed item. This is the default value for this parameter if it is not specified.

Remarks

An application can call the **CancelQueue** method passing a *CancelReqCount* value of 0, in order to retrieve the number of queued receive requests.

It is possible that a notification event is in process during this method call. If this occurs, the application may receive a notification following the successful completion of this function. Entering a *CancelReqCount* value that is larger than the queued requests will result in all requests being cancelled. No error will be returned under this condition. Calling this method while no queued requests are outstanding will result in a non-critical error return code indicating this condition.

CCSID Property

The **CCSID** property on an **IEGDataQueueCtrl** object indicates the character code set identifier (CCSID) that must match the data in the queue as represented on the remote host computer. This property affects how data conversion is handled using the Data Queue ActiveX Control. This property sets or returns a short value representing a host CCSID for the data file. The default value for this property is 37 representing a CCSID for U.S./Canada.

```
current CCSID = object.CCSID
object.CCSID = 37
```

Remarks

The **CCSID** property is used to set or return the character code set identifier (CCSID) matching the data in the data queue as represented on the remote host computer. This value is used for data conversion of any character data in the host data queue to the **PCCodePage** property specified representing ANSI or Unicode character data on the Windows machine.

ClearAll Method

The **ClearAll** method on the EIGDataQueue object removes all items from the queue.

```
object.ClearAll OverWrite
```

Parameters

OverWrite
This optional parameter specifies whether to overwrite data in the queue. This parameter can be set to one of the **eigAnswerYesNoEnum** constants shown in the table following the Parameters list.

Possible values for *OverWrite*

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the operation should overwrite data in the queue. This is the default value for this parameter if it is omitted from the method call.
eigAnswerNo	1	This value indicates that the operation should not overwrite data in the queue.

Connect Method

The **Connect** method on an **IEGDataQueueCtrl** object establishes a connection to the configured host using the Data Queue ActiveX Control and reports to the user an indication of the success or failure of the action.

```
object.Connect
```

Parameters

None.

Remarks

The **Connect** method is used to establish a connection to the host.

If the Data Queue ActiveX Control support DLL cannot be loaded, the Win32 **SetErrorMode** function is called with an error value of SEM_NOOPENFILEERRORBOX | SEM_FAILCRITICALERRORS. Other types of errors are returned using the **ISupportErrorInfo** object.

ConnectionState Property

The **ConnectionState** property on an **IEGDataQueueCtrl** object indicates the current state of the connection to the host using the Data Queue ActiveX Control. The state of a connection can be unspecified, idle, connecting, connected, or disconnecting. This property returns a Long value representing a **eigConnectionStateEnum**. The default value for this property is **eigConnStateIdle**.

```
currentConnectionState = object.ConnectionState
```

Remarks

The **ConnectionState** property is used to return the current state of the connection to the host. This property is read only and can be one of the following **eigConnectionStateEnum** constants:

Enumeration	Value	Description
eigConnStateUnspecified	-1	This value indicates that the connection state is unspecified.
eigConnStateIdle	0	This value indicates that the connection state is idle and no connection to the host exists.
eigConnStateConnecting	1	This value indicates that the connection state is in the process of connecting to the host.
eigConnStateConnected	2	This value indicates that the connection state is connected indicating a connection to the host.
eigConnStateDisconnecting	3	This value indicates that the connection state is in the process of disconnecting from the host.

ConnectionType Property

The **ConnectionType** property on an **IEGDataQueueCtrl** object indicates the network transport used for this connection. This property sets or returns a Long value representing a **eigConnectionTypeEnum**. The default value for this property is **eigConnTypeAPPC** indicating an APPC connection using SNA.

```
currentConnectionType = object.ConnectionType
```

Remarks

The **ConnectionType** property is used to set or return the connection type used to connect to the host. This property can be one of the following **eigConnectionTypeEnum** constants:

Enumeration	Value	Description
eigConnTypeUnspecified	-1	This value indicates that the connection type is unspecified.
eigConnTypeAPPC	0	This value indicates an APPC connection to the host using SNA LU 6.2.

If APPC (SNA LU 6.2) is selected for **ConnectionType**, then values for the **LocalLU**, **ModeName**, and **RemoteLU** properties are required.

CreateQueue Method

The **CreateQueue** method on an **IEGDataQueue** object creates a data queue using the Data Queue ActiveX Control. Following the successful creation of the queue, the object has a virtual connection to the data queue, such that a single instance of this object represents a single queue connection. In order to break this connection, the application can modify the **QueueName** property thus altering the queue association to a new value.

```
object.CreateQueue MaxMsgLength, QAuthority, QueueClass,
    AddSenderInfo, HostCCSID, InitialSize, queueLoc, recordLenCls,
    Title, AllowDupKeys, MakeKeyLen
```

Parameters

MaxMsgLength
This required parameter indicates the maximum length of a record in the queue represented as a short integer with possible values ranging from 1 to 31,744. This parameter defaults to a value of 256.

QAuthority
This required parameter specifies the authority to grant users of this queue. This parameter can be set to one of the **eigQAuthorityEnum** constants listed in the QAuthority Values table which follows the Parameters section.

QueueClass
This required parameter indicates how the data will be received from the data queue. This parameter can be set to one of the **eigQClassEnum** constants listed in the QueueClass Values table which follows the parameter section.

AddSenderInfo
This required parameter indicates whether the queue sender's ID should be kept. This parameter can be set to one of the **eigAnswerYesNoEnum** constants shown in the AddSenderInfo Values table which follows the Parameters section.

HostCCSID
This optional parameter indicates the character code set identifier (CCSID) represented as a short to be used on the host and affects how data conversion is handled. This value must match the data in the queue as represented on the remote host computer. This parameter has no default value, but if this parameter is not specified the *HostCCSID* defaults to the value of the **CCSID** property set on the **IEGDataQueueCtl** object.

InitialSize
This optional parameter indicates the initial data queue size represented as a short integer.

queueLoc
This optional parameter indicates the queue location represented as VARIANT_BOOL.

recordLenCls
This optional parameter indicates the record length class. This parameter can be set to one of the **eigRecordLenClsEnum** constants listed in the recordLenCls Values table which follows the Parameters section.

Title
This optional parameter indicates a text description of the queue represented as a BSTR with a maximum length of 50 characters. The default value for this parameter is an empty string.

AllowDupKeys
This optional parameter indicates whether duplicate keys are allowed. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the AllowDupKeys Values table which follows the Parameters section. Note that this parameter is only valid if the *QueueClass* is specified as **eigQClassKeyed**.

MakeKeyLen
This optional parameter indicates the maximum length of a key for this data queue represented as a short integer ranging from 1 to 256. Note that this parameter is only valid if the *QueueClass* is specified as **eigQClassKeyed**. This parameter has no default value.

***QAuthority* Values**

Enumeration	Value	Description
eigQAuthUnspecified	-1	This value indicates that the authority is unspecified.
eigQAuthDefault	0	This value indicates directory default authorization. This is the default value for this parameter if it is not specified.
eigQAuthAll	1	This value indicates all authorization.
eigQAuthExclude	2	This value indicates exclude authorization.
eigQAuthChange	3	This value indicates change authorization.

eigQAuthUse	4	This value indicates use authorization.
eigQAuthLibCreate	5	This value indicates library create authorization.

***AddSenderInfo* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the sender ID should be kept.
eigAnswerNo	1	This value indicates that the sender ID should not be kept. This is the default value for this parameter if it is not specified in the method call.

***QueueClass* Values**

Enumeration	Value	Description
eigQClassUnspecified	-1	
eigQClassFIFO	0	A first in, first out queue. This is the default value for this parameter if it is not specified.
eigQClassLIFO	1	A last in, first out queue.
eigQClassKeyed	2	A keyed ordered queue.

***recordLenCls* Values**

Enumeration	Value	Description
eigRecordLenUnspecified	-1	This value indicates that the record length class is unspecified.
eigRecordLenFixed	0	This value indicates fixed length records. This is the default value for this parameter if it is not specified.
eigRecordLenInitVarLen	1	This value indicates initial variable record length.
eigRecordLenVarLen	2	This value indicates variable record length.

***AllowDupKeys* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that duplicate key capability is supported.
eigAnswerNo	1	This value indicates that duplicate key capability is not supported. This is the default value for this parameter if it is not specified in the method call.

Remarks

The type of data queue created is dependent on the value of the *QueueClass* parameter. If a *QueueClass* of **eigQClassKeyed** is specified, then a keyed data queue is created. A *QueueClass* of **eigQClassFIFO** and **eigQClassLIFO** will result in a non-keyed data queue being created.

CreateQueueContainer Method

The **CreateQueue** method on an **IEGDataQueueCtl** object creates an instance of a **IEIGDataQueue** container object using the Data Queue ActiveX Control and optionally initializes the **QueueName** property. The default value for this property is VT_EMPTY.

```
object.CreateQueueContainer QueueName
```

Parameters

QueueName
This optional parameter is a BSTR string representing the name of the data queue that this object instance is connected to. This parameter corresponds with the value for the QueueName property.

Remarks

The created queue object is assumed to be associated with the connection object that created it for the life of the connection or the life of the queue object.

DeleteQueue Method

The **DeleteQueue** method on an **IEGDataQueue** object clears all messages from the queue and then deletes the queue using the Data Queue ActiveX Control.

```
object.DeleteQueue OverWriteData
```

Parameters

OverWriteData

This required parameter indicates whether data should be overwritten in the data queue. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the OverWriteData Values table which follows the Parameters section.

This parameter had a default value of **eigAnswerYes.**

***OverWriteData* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the operation should overwrite data in the queue. This is the default value for this parameter if it is omitted from the method call.
eigAnswerNo	1	This value indicates that the operation should not overwrite data in the queue.

Disconnect Method

The **Disconnect** method on an **IEGDataQueueCtrl** object terminates an existing connection to a host machine using the Data Queue ActiveX Control.

```
object.Disconnect
```

Parameters

None.

Remarks

The **Disconnect** method is used to terminate a connection to the host. Errors are returned using the **ISupportErrorInfo** object.

GetQueueItem Method

The **GetQueueItem** method on an **IEGDataQueue** object receives an item from a keyed queue.using the Data Queue ActiveX Control.

```
object.GetQueueItem QueueType, BlockComplete, PeekQItem,
    ProvideExtInfo, TimeOut, UserProfile, SenderInfo, SearchKey,
    SearchOrder, QueueItem
```

Parameters

QueueType
This required parameter specifies the type of the queue request. This parameter can be set to one of the **eigQItemTypeEnum** constants listed in the QueueType Values table which follows the Parameters section.

BlockComplete
This required parameter specifies whether the operation should block until the completion status is known. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the BlockComplete Values table which follows the parameters section.

PeekQItem
This required parameter indicates whether to keep the record in the data queue. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the PeekQItem Values table which follows the Parameters section.

ProvideExtInfo
This required parameter indicates whether to provide information in the External Job, name, and user properties. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the ProvideExtInfo Values table which follows the Parameters section.

TimeOut
This required parameter indicates the amount of time in seconds represented as a short value to block before indicating a failure. This parameter has a default value of 0.

UserProfile
This required parameter indicates whether to provide user profile feedback. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the UserProfile Values table which follows the Parameters section.

SenderInfo
This required parameter indicates whether the sender's ID should be returned. This parameter can be set to one of the **eigAnswerYesNoEnum** constants listed in the SenderInfo Values table which follows the Parameters section.

SearchKey
This optional parameter indicates the key used in conjunction with the *SearchOrder* parameter used to identify the queue data item being requested. This must be the same length as specified in the **CreateDataQueue** method call. This parameter is represented as a VARIANT and has no default value.

Note that this parameter is only valid when the *QueueClass* for the data queue is **eigQClassKeyed**.

SearchOrder
This optional parameter indicates the relational order used in receiving keyed data queue items. This parameter can be set to one of the **eigSearchKeyEnum** constants listed in the SearchOrder Values table which follows the Parameters section.

Note that this parameter is only valid when the *QueueClass* for the data queue is **eigQClassKeyed**.

QueueItem
This returned parameter is the requested **IEGDataQueueItem**.

***QueueType* Values**

Enumeration	Value	Description
eigKQItem	0	A keyed item.
eigQItem	1	A non-keyed item. This is the default value for this parameter if it is not specified.

***BlockComplete* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the operation should block until the completion status is known.

eigAnswerNo	1	This value indicates that the operation should not block. Although this is the default value for this parameter if it is omitted from the method call, this value is not supported by the Data Queue ActiveX Control. Asynchronous read operations are not supported.

PeekQItem Values

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the record should be kept in the data queue.
eigAnswerNo	1	This value indicates that the record should not be kept in the data queue. This is the default value for this parameter if it is omitted from the method call.

UserProfile Values

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that user profile feedback should be provided in the queue item.
eigAnswerNo	1	This value indicates that user profile feedback should not be provided in the queue item. This is the default value for this parameter if it is omitted from the method call.

***ProvideExtInfo* Values**

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that information should be provided in the External Job, name, and user properties on the **IEGDataQueueItem** to be returned.
eigAnswerNo	1	This value indicates that information should not be provided in the External Job, name, and user properties on the **IEGDataQueueItem** to be returned. This is the default value for this parameter if it is not specified in the method call.

SenderInfo Values

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the sender ID should be returned.
eigAnswerNo	1	This value indicates that the sender ID should not be returned. This is the default value for this parameter if it is not specified in the method call.

SearchOrder Values

Enumeration	Value	Description
eigSearchKeyUnspecified	-1	This value indicates that the *SearchOrder* parameter is unspecified.
eigSearchKeyEqual	0	This value indicates a search for items equal to the specified *SearchKey* parameter.
eigSearchKeyGreaterThan	1	This value indicates a search for items greater than the specified *SearchKey* parameter.
eigSearchKeyLessThan	2	This value indicates a search for items less than the specified *SearchKey* parameter.
eigSearchKeyGreaterEqual	3	This value indicates a search for items greater than or equal to the specified *SearchKey* parameter.
eigSearchKeyLessEqual	4	This value indicates a search for items less than or equal to the specified *SearchKey* parameter.

Remarks

If the *BlockComplete* parameter is **eigAnswerNo** and no queue item is available, the request will be queued. Following the receipt of a queue data item, an event will be fired to the client indicating the availability of data. The client application will be

required to call this function again in order to receive the queue data.

If the *BlockComplete* parameter is **eigAnswerYes** and no data arrives on the queue within the specified amount of time, the operation is cancelled and no data is returned. An error indication is returned to the client indicating that a timeout has occurred.

Each call to this method may result in a queued process. Multiple calls may result in multiple notifications.

If the *SenderInfo* item is empty, then the information will not be returned. The client must allocate the storage as an indication that it wishes to receive this information. The type of data queue is dependent on the value of the *QueueClass* parameter when the data queue is created. For a *QueueClass* of **eigQClassKeyed**, a keyed data queue is created. A *QueueClass* of **eigQClassFIFO** or **eigQClassLIFO** will result in a non-keyed data queue being created.

LocalLU Property

The **LocalLU** property on an **IEGDataQueueCtrl** object indicates the local LU alias for an APPC (SNA LU 6.2) connection type to the remote host computer using the Data Queue ActiveX Control. This property sets or returns a BSTR string value representing the local LU name. The default value for this property is the "LOCAL" string.

```
currentLocalLu = object.LocalLU
object.LocalLu = "Local2"
```

Remarks

The **LocalLU** property is used to set or return the local LU alias. When LU 6.2 (SNA) is selected for the ConnectionType property, this property must match the name of the local LU alias configured using SNA Manager.

ModeName Property

The **ModeName** property on an **IEGDataQueueCtrl** object indicates the APPC mode used for an APPC (SNA LU 6.2) connection type to the remote host computer using the Data Queue ActiveX Control. This property sets or returns a BSTR string value representing the APPC mode. The default value for this property is the "QPCSUPP" string.

```
currentAppcMode = object.ModeName
object.ModeName = "QPCSUPP"
```

Remarks

The **ModeName** property is used to set or return the APPC mode. When APPC (LU 6.2 SNA) is selected for the ConnectionType property, this field must bet set to the APPC mode that matches the host configuration and Host Integration Server configuration. This property is ignored when TCP/IP is selected for the **ConnectionType** property.

Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security).

Password Property

The **Password** property on an **IEGDataQueueCtrl** object indicates the password used for authentication. This property affects connection and authentication to a host computer using the Data Queue ActiveX Control. This property sets or returns a BSTR and has no default value.

```
currentUserPassword = object.Password
object.Password = Dialog.UserPassword
```

Remarks

The **Password** property is used to set or return the password used for authenticating the user on a host computer. A valid user name and password are normally required to access data on a host computer. The password is case sensitive and is normally displayed as asterisks in a dialog box for security purposes.

PCCodePage Property

The **PCCodePage** property on an **IEGDataQueueCtrl** object indicates the code page to be used on the PC. This property affects how data conversion is handled using the Data Queue ActiveX Control. This property sets or returns a short value representing the PC code page for the data file. The default value for this property is 1252 representing a PC code page of Latin 1.

```
currentPCCodePage = object.PCCodePage
object.PCCodePage = 1252
```

Remarks

The **PCCodePage** property is used to set or return the code page to be used on the PC. This value is used for data conversion of any character data in the host file to ANSI or Unicode character data in the local file on the Windows machine.

QueryAttribute Method

The **QueryAttribute** method on the EIGDataQueue object returns the value of an attribute associated with a data queue.

```
object.QueryAttribute Attribute, Value
```

Parameters

Attribute
This required parameter indicates the attribute value to be retrieved. This parameter can be set to one of the **eigAttributeEnum** constants listed in the Attribute Values table which follows the Parameters section.

Value
This required parameter points to a variant that will receive the value for this attribute.

Attribute Values

Enumeration	Value	Description
eigAttributeUnspecified	-1	
eigAttributeCCSID	0	The code character set identifier used on the host.
eigAttributeDirName	1	Directory name
eigAttributeDataClass	2	Data class name
eigAttributeKeyDef	3	Key definition
eigAttributeDupKeys	4	Indicates whether the duplicate keys capability is enabled for this data queue.
eigAttributeMgmCls	5	Management class name
eigAttributeQueCls	6	The queue class corresponding with the *QueueClass* parameter on the **CreateQueue** method.
eigAttributeSize	7	The queue initial size
eigAttributeQLoc	8	Queue location
eigAttributeSenderInfo	9	Queue senders ID kept
eigAttributeMaxMsgLen	10	The maximum record length for a message.
eigAttributeRecLenClass	11	Record length class
eigAttributeStgClass	12	Storage class name
eigAttributeTitle	13	Description text

Remarks

Most of these attributes correspond with the parameters specified to the CreateQueue method when the data queue is created.

QueueName Property

The **QueueName** property on an **IEGDataQueue** object indicates the name of the data queue this object is associated with.

```
currentQueueName = object.QueueName
object.QueueName = "OrderEntry"
```

Remarks

The value of the **QueueName** property is the name of the queue represented as a BSTR string.

Following the successful creation of a queue using the CreateQueue method, the object has a virtual connection to the data queue, such that a single instance of this object represents a single queue connection. In order to break this connection, a client application can modify the **QueueName** property thus altering the queue association to a new value.

RemoteLU Property

The **RemoteLU** property on an **IEGDataQueueCtrl** object indicates the remote LU alias for an APPC (SNA) connection type to the remote host computer using the Data Queue ActiveX Control. This property sets or returns a BSTR string value representing the remote LU name. This property has no default value.

```
currentRemoteLu = object.RemoteLU
object.RemoteLu = "Remote10"
```

Remarks

The **RemoteLU** property is used to set or return the remote LU alias. When LU 6.2 (SNA) is selected for the ConnectionType property, this property must match the name of the remote LU alias configured using SNA Manager.

SetAttribute Method

The **SetAttribute** method on the EIGDataQueue object changes an attribute associated with a data queue.

```
object.SetAttribute Attribute, Value
```

Parameters

Attribute
This required parameter indicates the attribute to be set. This parameter can be set to one of the **eigAttributeEnum** constants listed in the Attribute Values table which follows the Parameters section.

Value
This required parameter specifies a variant representing the value to set for this attribute.

Attribute Values

Enumeration	Value	Description
eigAttributeUnspecified	-1	
eigAttributeCCSID	0	Coded character set identifier
eigAttributeDirName	1	Directory name
eigAttributeDataClass	2	Data class name
eigAttributeKeyDef	3	Key definition
eigAttributeDupKeys	4	Duplicate keys capability
eigAttributeMgmCls	5	Management class name
eigAttributeQueCls	6	The queue class corresponding with the *QueueClass* parameter on the **CreateQueue** method.
eigAttributeSize	7	Queue initial size
eigAttributeQLoc	8	Queue location
eigAttributeSenderInfo	9	Queue senders ID kept
eigAttributeMaxMsgLen	10	Record length
eigAttributeRecLenClass	11	Record length class
eigAttributeStgClass	12	Storage class name
eigAttributeTitle	13	Description text

Remarks

Most of these attributes correspond with the parameters specified to the CreateQueue method when the data queue is created.

StopQueue Method

The **StopQueue** method on an **IEGDataQueue** object suspend send and receive operations for a queue using the Data Queue ActiveX Control.

```
object.StopQueue
```

Parameters

None.

UserID Property

The **UserID** property on an **IEGDataQueueCtrl** object indicates the username used for authentication. This property affects connection and authentication to a host computer using the Data Queue ActiveX Control. This property sets or returns a BSTR and has no default value.

```
currentUserName = object.UserID
object.UserID = Dialog.UserName
```

Remarks

The **UserID** property is used to set or return the username used for authenticating the user on a host computer. A valid user name and password are normally required to access data on a host computer. The username is case sensitive.

Host File Transfer ActiveX Control Reference

This section provides reference information on specific ActiveX® methods, properties, and event notifications supported by the Microsoft® Host File Transfer ActiveX Control. The function syntax and code examples are based on Microsoft Visual Basic.

AppendToEnd Property

The **AppendToEnd** property on an **IEIGFileTransferCtl** object indicates whether a file transfer operation should append to the end of a file if the file exists, or whether a file transfer operation should overwrite the existing contents replacing the data with the new information. This property affects file transfer operations using the Host File Transfer ActiveX Control.This property sets or returns a Long value representing a **eigAnswerYesNoEnum**. The default value for this property is **eigAnswerYes**.

```
currentAppendFlag = object.AppendToEnd
object.AppendToEnd = eigAnswerYes
```

Remarks

The **AppendToEnd** property is used to set or return a flag that indicates whether a file transfer operation will append to the end of a file or overwrite the file. This property can be set to one of the following **eigAnswerYesNoEnum** constants:

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the file transfer operation will append to the end of a file if it exists.
eigAnswerNo	1	This value indicates that the file transfer operation will overwrite the existing contents of a file if it exists.

The **AppendToEnd** property and the **OverwriteHostFile** property are mutually exclusive, so it is not possible to enable (set to yes) one of these properties before the opposing property is disabled (set to no). The **AppendToEnd** property takes precedence over the **OverwriteHostFile** property, since **AppendToEnd** defaults to yes and **OverwriteHostFile** defaults to no. Consequently, the order that these properties are set will affect the outcome. For example, the following order will result in the properties being set correctly:

```
FileTransfer.AppendToEnd = eigAnswerNo         // correctly set to no
FileTransfer.OverwriteHostFile = eigAnswerYes  // correctly set to yes
```

In contrast, setting the properties in the improper order will cause the properties to be set incorrectly as follows:

```
FileTransfer.OverwriteHostFile = eigAnswerYes   // remains at no
// AppendToEnd defaults to eigAnswerYes, so this change is illegal
FileTransfer.AppendToEnd = eigAnswerNo          // correctly set to no
```

In this second case, the **OverwriteHostFile** property cannot be set to yes (enabled) until **AppendToEnd** property is set to no (disabled).

Cancel Method

The **Cancel** method on an **IEIGFileTransferCtl** object terminates a file transfer operation that is already in progress. This method cancels a file transfer using the Host File Transfer ActiveX Control.

```
object.Cancel
```

Parameters

None.

Remarks

The **Cancel** method is used to cancel a file transfer operation that is already in progress. If the **Cancel** method is executed while uploading a file with the **AppendToEnd** property set to yes, this will result in no change to the host file. However, if the **Cancel** method is executed while uploading a file with the **OverwriteHostFile** property set to yes, this will result in an empty host file. The **Cancel** method implies the transfer has been stopped and all the files are at their original values but this is not really the case when the **OverwriteHostFile** property is set to yes.

CCSID Property

The **CCSID** property on an **IEIGFileTransferCtl** object indicates the character code set identifier (CCSID) that must match the data in the file as represented on the remote host computer. This property affects how data conversion is handled using the Host File Transfer ActiveX Control. This property sets or returns a short value representing a host CCSID for the data file. The default value for this property is 37 representing a CCSID for U.S./Canada.

```
current CCSID = object.CCSID
object.CCSID = 37
```

Remarks

The **CCSID** property is used to set or return the character code set identifier (CCSID) matching the data in the file as represented on the remote host computer. This value is used for data conversion of any character data in the host file to the **PCCodePage** property specified representing ANSI or Unicode character data on the Windows computer.

Connect Method

The **Connect** method on an **IEIGFileTransferCtl** object establishes a connection to the configured host using the Host File Transfer ActiveX Control and reports to the user an indication of the success or failure of the action.

```
object.Connect
```

Parameters

None.

Remarks

The **Connect** method is used to establish a connection to the host.

If the Host File Transfer Control support DLL cannot be loaded, the Win32 **SetErrorMode** function is called with an error value of SEM_NOOPENFILEERRORBOX | SEM_FAILCRITICALERRORS. Other types of errors are returned using the **ISupportErrorInfo** object.

ConnectionState Property

The **ConnectionState** property on an **IEIGFileTransferCtl** object indicates the current state of the connection to the host using the Host File Transfer ActiveX Control. The state of a connection can be unspecified, idle, connecting, connected, or disconnecting. This property returns a Long value representing a **eigConnectionStateEnum**. The default value for this property is **eigConnStateIdle**.

```
currentConnectionState = object.ConnectionState
```

Remarks

The **ConnectionState** property is used to return the current state of the connection to the host. This property is read only and can be one of the following **eigConnectionStateEnum** constants:

Enumeration	Value	Description
eigConnStateUnspecified	-1	This value indicates that the connection state is unspecified.
eigConnStateIdle	0	This value indicates that the connection state is idle and no connection to the host exists.
eigConnStateConnecting	1	This value indicates that the connection state is in the process of connecting to the host.
eigConnStateConnected	2	This value indicates that the connection state is connected indicating a connection to the host.
eigConnStateDisconnecting	3	This value indicates that the connection state is in the process of disconnecting from the host.

ConnectionType Property

The **ConnectionType** property on an **IEIGFileTransferCtl** object indicates the network transport used for this connection. The **ConnectioneType** property designates whether the Host File Transfer ActiveX Control connects via APPC (SNA LU6.2) or TCP/IP. This property sets or returns a Long value representing a **eigConnectionTypeEnum**. The default value for this property is **eigConnTypeAPPC** indicating an APPC connection using SNA.

```
currentConnectionType = object.ConnectionType
object.ConnectionType = eigConnTypeTCPIP
```

Remarks

The **ConnectionType** property is used to set or return the connection type used to connect to the host. This property can be one of the following **eigConnectionTypeEnum** constants:

Enumeration	Value	Description
eigConnTypeUnspecified	-1	This value indicates that the connection type is unspecified.
eigConnTypeAPPC	0	This value indicates an APPC connection to the host using SNA LU 6.2.
eigConnTypeTCPIP	1	This value indicates a TCP/IP connection to the host.

If APPC (SNA) is selected for **ConnectionType**, then values for the **LocalLU**, **ModeName**, and **RemoteLU** properties are required.

If TCP/IP is selected for **ConnectionType**, then values for the **NetAddr** and **NetPort** properties are required.

CreateIfNonExisting Property

The **CreateIfNonExisting** property on a **IEIGFileTransferCtl** object indicates whether a file transfer operation should create a new destination file if one does not already exist. This property affects file transfer operations using the Host File Transfer ActiveX Control. This property sets or returns a Long value representing a **eigAnswerYesNoEnum**. The default value for this property is **eigAnswerNo**.

```
currentCreateFlag = object.CreateIfNonExisting
object.CreateIfNonExisting = eigAnswerYes
```

Remarks

The **CreateIfNonExisting** property is used to set or return a flag that indicates whether a file transfer operation should create a new destination file if one does not already exist. This property can be set to one of the following **eigAnswerYesNoEnum** constants:

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the file transfer operation will create a new destination file if the file does not already exist.
eigAnswerNo	1	This value indicates that the file transfer operation will not create a new destination file if the file does not already exi st. If the destination file does not already exist, then the file copy operation will not take place.

Disconnect Method

The **Disconnect** method on an **IEIGFileTransferCtl** object terminates an existing connection to a host machine using the Host File Transfer ActiveX Control.

```
object.Disconnect
```

Parameters

None.

Remarks

The **Disconnect** method is used to terminate a connection to the host. Errors are returned using the **ISupportErrorInfo** object.

GetFile Method

The **GetFile** method on an **IEIGFileTransferCtl** object copies a file from host storage to local storage using the Host File Transfer ActiveX Control.

```
object.GetFile LocalFile, HostFile
```

Parameters

LocalFile
This required parameter specifies the path of a local file that will be written to as a result of this operation.
HostFile
This required parameter specifies the name of the host file that will be copied to the local file.

Remarks

The **GetFile** method can only be called after a connection has been established to the host (when the ConnectionState property is connected). The behavior of the **GetFile** method is affected by the values of the AppendToEnd, CreateIfNonExisting, and OverwriteHostFile properties.

Errors are returned for this method using the **ISupportErrorInfo** object.

LocalLU Property

The **LocalLU** property on an **IEIGFileTransferCtl** object indicates the local LU alias for an APPC (SNA) connection type to the remote host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value representing the local LU name. The default value for this property is the "LOCAL" string.

```
currentLocalLu = object.LocalLU
object.LocalLu = "Local2"
```

Remarks

The **LocalLU** property is used to set or return the local LU alias. When LU 6.2 (SNA) is selected for the ConnectionType property, this property must match the name of the local LU alias configured using SNA Manager.

This property is ignored when TCP/IP is selected for the **ConnectionType** property.

ModeName Property

The **ModeName** property on an **IEIGFileTransferCtl** object indicates the APPC mode used for an APPC (SNA) connection type to the remote host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value representing the APPC mode. The default value for this property is the "QPCSUPP" string.

```
currentAppcMode = object.ModeName
object.ModeName = "QPCSUPP"
```

Remarks

The **ModeName** property is used to set or return the APPC mode. When APPC (LU 6.2 SNA) is selected for the ConnectionType property, this field must bet set to the APPC mode that matches the host configuration and Host Integration Server configuration. This property is ignored when TCP/IP is selected for the **ConnectionType** property.

Legal values for the APPC mode include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security).

NetAddr Property

The **NetAddr** property on an **IEIGFileTransferCtl** object indicates the IP address of the host computer for a TCP/IP connection type to the remote host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value representing the IP address of the host computer. This property has no default value.

```
currentHostIP = object.NetAddr
object.NetAddr = "207.136.131.30"
```

Remarks

The **NetAddr** property is used to set or return the IP address of the host computer. When TCP/IP is selected for the ConnectionType property, this property must match the IP address of the host computer used where files will be transferred. This property can be an IP address or the name representing the host IP address using the Domain Name System (sna.microsoft.com, for example). This property is ignored when SNA is selected for the **ConnectionType** property.

NetPort Property

The **NetPort** property on an **IEIGFileTransferCtl** object indicates the TCP/IP port used for communication with the host for a TCP/IP connection type to the remote host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value representing the TCP/IP port used for communication with the host. The default value for this property is the string "446" representing TCP/IP port 446.

```
currentIPPort = object.NetPort
object.NetPort = "446"
```

Remarks

The **NetPort** property is used to set or return the TCP/IP port used for communication with the host. When TCP/IP has been selected for the ConnectionType property, this parameter is the TCP/IP port used for communication with the host. This property is ignored when SNA is selected for the **ConnectionType** property.

OverwriteHostFile Property

The **OverwriteHostFile** property on a **IEIGFileTransferCtl** object indicates whether a file transfer operation request to copy a file that will write over an existing file will be executed. This property affects file transfer operations using the Host File Transfer ActiveX Control.This property sets or returns a Long value representing a **eigAnswerYesNoEnum**. The default value for this property is **eigAnswerNo**.

```
currentOverwriteFlag = object.OverwriteHostFile
object.OverwriteHostFile = eigAnswerYes
```

Remarks

The **OverwriteHostFile** property is used to set or return a flag that indicates whether a file transfer operation will over an existing file. This property can be set to one of the following **eigAnswerYesNoEnum** constants:

Enumeration	Value	Description
eigAnswerYes	0	This value indicates that the file transfer operation will overwrite an existing host file if it exists.
eigAnswerNo	1	This value indicates that the file transfer operation will not overwrite and existing host file file if it exists.

The **OverwriteHostFile** property and the **AppendToEnd** property are mutually exclusive, so it is not possible to enable (set to yes) one of these properties before the opposing property is disabled (set to no). The **AppendToEnd** property takes precedence over the **OverwriteHostFile** property, since **AppendToEnd**defaults to yes and **OverwriteHostFile** defaults to no. Consequently, the order that these properties are set will affect the outcome. For example, the following order will result in the properties being set correctly:

```
FileTransfer.AppendToEnd = eigAnswerNo         // correctly set to no
FileTransfer.OverwriteHostFile = eigAnswerYes  // correctly set to yes
```

In contrast, setting the properties in the improper order will cause the properties to be set incorrectly as follows:

```
FileTransfer.OverwriteHostFile = eigAnswerYes   // remains at no
// AppendToEnd defaults to eigAnswerYes, so this change is illegal
FileTransfer.AppendToEnd = eigAnswerNo          // correctly set to no
```

In this second case, the **OverwriteHostFile** property cannot be set to yes (enabled) until **AppendToEnd** property is set to no (disabled).

Password Property

The **Password** property on an **IEIGFileTransferCtl** object indicates the password used for authentication. This property affects connection and authentication to a host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR and has no default value.

```
currentUserPassword = object.Password
object.Password = Dialog.UserPassword
```

Remarks

The **Password** property is used to set or return the password used for authenticating the user on a host computer. A valid user name and password are normally required to access files on a host computer. The password is case sensitive and is normally displayed as asterisks in a dialog box for security purposes.

PCCodePage Property

The **PCCodePage** property on an **IEIGFileTransferCtl** object indicates the code page to be used on the PC. This property affects how data conversion is handled using the Host File Transfer ActiveX Control. This property sets or returns a short value representing the PC code page for the data file. The default value for this property is 1252 representing a PC code page of Latin 1.

```
currentPCCodePage = object.PCCodePage
object.PCCodePage = 1252
```

Remarks

The **PCCodePage** property is used to set or return the code page to be used on the PC. This value is used for data conversion of any character data in the host file to ANSI or Unicode character data in the local file on the Windows machine.

PutFile Method

The **PutFile** method on an **IEIGFileTransferCtl** object copies a file from host storage to local storage using the Host File Transfer ActiveX Control.

```
object.PutFile HostFile, LocalFile
```

Parameters

HostFile
This required parameter specifies the name of the host file that will be written to as a result of this operation. This parameter is a BSTR.

LocalFile
This required parameter specifies the path to a local file that will be copied to the host file. This parameter is a BSTR.

Remarks

The **PutFile** method can only be called after a connection has been established to the host (when the ConnectionState property is connected). The behavior of the **PutFile** method is affected by the values of the AppendToEnd, CreateIfNonExisting, and OverwriteHostFile properties.

Errors are returned for this method using the **ISupportErrorInfo** object.

RDBName Property

The **RDBName** property on an **IEIGFileTransferCtl** object indicates the remote database name and the Host Column Description (HCD) file that describes the data types and data conversions used to transfer this file using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value and has no default value.

```
currentRDbName = object.RDBName
object.RDBName = "Inventory"
```

Remarks

The **RDBName** property is used to set or return the name of the remote database name and the Host Column Description (HCD) file that describes the data types and data conversions used to transfer this file. The HCD file describing the data should be located in the system subdirectory below the root directory where Host Integration Server was installed. Setup defaults to the following location: C:\Program Files\Host Integration Server

When TCP/IP is selected for the **ConnectionType** property, the **RDBName** must also match the name of the remote database system.

RemoteLU Property

The **RemoteLU** property on an **IEIGFileTransferCtl** object indicates the remote LU alias for an APPC (SNA) connection type to the remote host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR string value representing the remote LU name. This property has no default value.

```
currentRemoteLu = object.RemoteLU
object.RemoteLu = "Remote10"
```

Remarks

The **RemoteLU** property is used to set or return the remote LU alias. When LU 6.2 (SNA) is selected for the ConnectionType property, this property must match the name of the remote LU alias configured using SNA Manager.

This property is ignored when TCP/IP is selected for the **ConnectionType** property.

UserID Property

The **UserID** property on an **IEIGFileTransferCtl** object indicates the username used for authentication. This property affects connection and authentication to a host computer using the Host File Transfer ActiveX Control. This property sets or returns a BSTR and has no default value.

```
currentUserName = object.UserID
object.UserID = Dialog.UserName
```

Remarks

The **UserID** property is used to set or return the username used for authenticating the user on a host computer. A valid user name and password are normally required to access files on a host computer. The username is case sensitive.

Host Column Description

The Microsoft® OLE DB Provider for AS/400 and VSAM uses a host column description (HCD) file to specify how data on the host is converted by the OLE DB provider. These HCD files are not necessary when used with AS/400 machines since the host data format is automatically determined by the OLE DB Provider. When used with AS/400 machines, the OLE DB Provider uses default conversions from host data type to OLE DB data types. However, HCD files can be used with AS/400 machines to override the host data format and specify a particular local OLE DB data type that the data is to be converted to.

The following topics describe the host column description file format in detail and provide an example of an HCD file for illustration.

Host Column Description File Format

The Microsoft® OLE DB Provider for AS/400 and VSAM uses a host column description (HCD) file to describe the format of data files on IBM mainframes and dictate how the data in columns or fields in these files is to be converted by the OLE DB provider. Host column description files can also be used for data on AS/400 machines to override the data format description maintained by the AS/400 host.

The default naming convention for the HCD file is <*data source*>.hcd, where <*data source*> is the remote LU alias, and hcd is the file extension of the record description file. For example, if the remote LU alias configured on Host Integration Server is named ABC01234, the host column description file would be named ABC01234.hcd as the default. Any name may be used for an HCD file as long as the file extension is ".hcd."

All the VSAM files that have local record descriptions are listed in the [Files] section of the record description file. Each file that is listed in the [Files] section has a record description section, and that section name is the same as the file name. For example, a VSAM file named PUBS/AUTHORS has its record description saved in the [PUBS/AUTHORS] section.

The first key of a record description section is the number of the columns for the file. The syntax is as follows:

```
numcol=<number of columns>
```

where <*number of columns*> is the actual number of columns described in the section. Each column of a file is described by one key. The naming convention for the key name is as follows:

```
col<column number>
```

For example the ninth column's key is **col9**. Each key has an attribute list that contains eleven concatenated attributes that describe the column. Attributes are separated by semicolons.

Host Column Description Attributes

Attributes are shown in the following table:

Attribute	Field number	Description
Reserved	1	Always zero
Column Name	2	Character string
Alias	3	Character string
Precision	4	Numeric value
Scale	5	Numeric value
Length	6	Numeric value
Host Type	7	Keyword representing the host data type
OLE DB Type	8	Keyword representing the local OLE DB data type
Nullable	9	Y or N
CCSID	10	Numeric value
Title	11	Character string

Column Name

The column name attribute is the character string that represents the name of the column. This attribute may be null.

Alias

The alias attribute is an optional character string that represents an alias label for the column string name. This attribute may be null.

Precision

The precision is the total number of decimal digits in the column containing numeric data on the host. The only two fixed-point numeric data types that require this information are the NUMERIC and DECIMAL keyword data types, and for these types this field cannot be null and must contain a valid numeric value.

The precision must be set the same as the length attribute for CHAR and BINARY keyword data types, and set to zero for the other types. (Note that under ODBC, the precision was also used to indicate the length of nonnumeric data types including character, date, time, and binary data types.) Precision has no default value and must not be left null.

Scale

The scale is the number of decimal digits to the right of any decimal point for numeric data on the host. The only two numeric data types that require this information are the NUMERIC and DECIMAL keyword data types, and for these types this field cannot be null and must contain a valid numeric value. For other numeric (the SINGLE and DOUBLE keywords, for example) and nonnumeric data types (binary, character, date, time, and timestamp), the scale should be set to zero. This field must not be left null and must contain a numeric value.

Length

The length attribute is the total length of the data on the host. This field must not be left empty and must contain a numeric value.

Host Type

The Host Type attribute is a keyword value that represents the data type of the host data. This keyword value is based on standard data types used on AS/400 and VSAM files. If no keyword is entered, this attribute defaults to the BINARY keyword. The allowable types are as follows:

Host type name	Keyword	Comment
Binary	BINARY	Fixed-length binary data (no character code conversions). The length must be specified.
Character	CHAR	Fixed-length character data. The length must be specified.
Date	DATE	The date represented as character data in the format yyyy-mm-dd that fits in 10 bytes
DBCS	DBCS	Fixed-length character data that can contain only DBCS data.
DCBS – Mixed Either	MIXED_EITHER	Fixed-length character data that can contain either DBCS or alphanumeric data.
DCBS – Mixed Open	MIXED_OPEN	Fixed-length character data that can contain both DBCS and alphanumeric data. DBCS data is distinguished from alphanumeric data with shift-control characters.
DBCS – Variable	VARDBCS	Variable-length character data with a prefix of 2 bytes for length that contains only DBCS data. The maximum possible length for the column containing this data type must be specified.

DCBS – Variable Mixed Either	VARMIXED_EITHER	Variable-length character data with a prefix of 2 bytes for length that can contain either DBCS or alphanumeric data. The maximum possible length for the column containing this data type must be specified.
DCBS – Variable Mixed Open	VARMIXED_OPEN	Variable-length character data with a prefix of 2 bytes for length that can contain both DBCS and alphanumeric data. DBCS data is distinguished from alphanumeric data with shift-control characters. The maximum possible length for the column containing this data type must be specified.
Double	DOUBLE	Floating-point data that fits in 8 bytes (64 bits).
Long	LONG	Integer data that fits in 4 bytes (32 bits).
Long Variable Binary	LONGVARBINARY	Variable-length binary data represented as an unsigned character array with prefix 2 bytes for length. The maximum possible length for the column containing this data type must be specified.
Long Variable Character	LONGVARCHAR	Variable-length character data with a prefix of 2 bytes for the length. The maximum possible length for the column containing this data type must be specified.
Packed	PACKED	Packed decimal numeric data where the precision, and scale are exactly as specified.
Short	SHORT	Integer data that fits in 2 bytes (16 bits).
Single	SINGLE	Floating-point data that fits in 4 bytes (32 bits).
Time	TIME	The time represented as character data in the format hh:mm:ss that fits in 8 bytes.
Time Stamp	TIMESTAMP	Timestamp represented as characters in the format yyyy-mm-dd hh:mm:ss.ffffff that fits in 19 bytes.
Variable Binary	VARBINARY	Variable-length binary data represented as an unsigned character array with prefix 2 bytes for length. The maximum possible length for the column containing this data type must be specified.
Variable Character	VARCHAR	Variable-length character data with a prefix of 2 bytes for length. The maximum possible length for the column containing this data type must be specified.
Zoned	ZONED	Zoned decimal numeric data where the precision and scale are exactly as specified.

Note that the OLE DB provider limits the maximum length character field that can be accessed on an AS/400 machine to 32,745. Attempting to access a character field greater than this length on an AS/400 machine can have unpredictable results and can cause the OLE DB Provider to hang.

Note that the floating-point data format assumed by the OLE DB Provider for AS/400 and VSAM depends on the host. For AS/400, the host floating-point data format is assumed to be IEEE. On mainframe hosts, floating-point data types are assumed to be in IBM floating-point formats. Since OLE DB supports the IEEE floating-point format, data conversion errors can occur when converting the extreme values of VSAM floating-point data in IBM format to IEEE floating-point data by the OLE DB provider.

Note that the DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, and SMALLINT keywords should not be used in HCD files and should be replaced with the newer keywords as follows:

The DOUBLE keyword replaces the FLOAT keyword.

The LONG keyword replaces the INTEGER keyword.

The PACKED keyword replaces the NUMERIC keyword.

The SHORT keyword replaces the SMALLINT keyword.

The SINGLE replaces the REAL keyword.

The ZONED keyword replaces the DECIMAL keyword.

The Data Descriptions management console snap-in provided with Host Integration Server will not work properly with HCD files containing these older keywords and will give unpredictable results.

OLE DB Type

The OLE DB Type attribute is a keyword that represents the data type of the local PC data. This keyword value is based on the standard OLE DB data types as defined in the OLEDB.H header file included with the OLE DB SDK version 1.5 and later The data structures for the date, time, and timestamp C data types are defined as typedefs in the OLEDB.H header file included with the OLE DB SDK. Similar data types are also used by ODBC. If no keyword is entered, this attribute defaults to the BINARY keyword.

For the decimal and numeric host data types, the OLE DB Type attribute must be set to DBTYPE_STR. These two fixed-point numeric types are currently converted to character data strings by the DDM layer of the Microsoft OLE DB Provider for AS/400 and VSAM. If these host types are converted to any of the defined OLE DB numeric C types, then numeric accuracy can be lost.

The allowable types are as follows:

OLE DB type name	Keyword	Comment
DBTYPE_BYTES	BINARY	Fixed-length binary data represented as an unsigned char array.
DBTYPE_DBDATE	DATE	The OLE DB DBDATE typedef struct as defined in the OLEDB.H header file.
DBTYPE_DBTIME	TIME	The OLE DB DBTIME typedef as defined in the OLEDB.H header file.
DBTYPE_DBTIMESTAMP	TIMESTAMP	The OLE DB DBTIMESTAMP typedef struct as defined in the OLEDB.H header file.
DBTYPE_DECIMAL	DECIMAL	The OLE DB DECIMAL typedef struct as defined in the OLEDB.H header file.
DBTYPE_I2	SHORT	Integer data stored in 2 bytes (16 bits).
DBTYPE_I4	LONG	Integer data stored in 4 bytes (32 bits).
DBTYPE_NUMERIC	NUMERIC	The OLE DB NUMERIC typedef struct as defined in the OLEDB.H header file.
DBTYPE_R4	FLOAT	Floating-point data stored in 4 bytes (32 bits).
DBTYPE_R8	DOUBLE	Floating-point data stored in 8 bytes (64 bits).
DBTYPE_STR	CHAR	Character data.
DBTYPE_I1		Not supported (signed tiny integer stored in one byte, 8 bits).
DBTYPE_I8		Not supported (signed long long integer stored in eight bytes, 64 bits).
DBTYPE_UI1		Not supported (unsigned tiny integer stored in one byte, 8 bits).
DBTYPE_UI2		Not supported (unsigned short integer stored in two bytes, 16 bits).
DBTYPE_UI4		Not supported (unsigned long integer stored in four bytes, 32 bits).
DBTYPE_UI8		Not supported (unsigned long long integer stored in eight bytes, 64 bits).

Nullable

The Nullable attribute indicates whether this field can be a null value. Legal values for this field are Y or N. If this field is empty, the default value for nullable is N. The current version of the OLE DB Provider for AS/400 and VSAM does support nullable fields, so this value must be set to N.

CCSID

The Code Character Set Identifier (CCSID) attribute indicates the character set used on the host. If this field is empty, the default value for CCSID is set to EBCDIC US English (37). The CCSID setting used by the OLE DB Provider must be set to match the CCSID actually used on the host, otherwise data loss will occur. Note that some AS/400 systems default to a CCSID of 937, rather than 37, for enabling double-byte character sets (DBCS).

If the CCSID is set to 0 or 65535, then no character translation will take place when converting character data from the host to the PC by the OLE DB Provider. The allowable values for CCSID when used with OLE DB Provider for AS/400 and VSAM are as follows:

EBCDIC character set	CCSID value
U.S./Canada	37
Germany	273
Denmark/Norway	277
Finland/Sweden	278
Italy	280
Latin America/Spain	284
United Kingdom	285
France	297
International	500

Note that this value needs to correspond to the CCSID used on the host.

Title

The Title attribute is an optional character string that represents a comment describing the column. This attribute may be null.

Host Column Description Example File

The following is an example for an HCD file containing two sample database file descriptions — SAMPLE/ACCOUNTS and PUBS/AUTHORS. The first sample file (SAMPLE/ACCOUNTS) has four columns of data while the second example (PUBS/AUTHORS) has nine columns that must be described.

```
[files]
SAMPLE/ACCOUNTS=1
PUBS/AUTHORS=1

[SAMPLE/ACCOUNTS]
numcol=4
col1=0;CUST_NO;CUST_NO;8;0;0;ZONED;LONG;N;37;;
col2=0;CUST_NAME;CUST_NAME;0;0;40;CHAR;CHAR;N;37;;
col3=0;BALANCE;BALANCE;10;2;0;ZONED;FLOAT;N;37;;
col4=0;LAST_ACC;LAST_ACC;0;0;26;TIMESTAMP;TIMESTAMP;N;37;;

[PUBS/AUTHORS]
NumCol=9
Col1=0;AU_ID;AU_ID;11;0;11;CHAR;CHAR;N;37;;
Col2=0;AU_LNAME;AU_LNAME;0;0;40;VARCHAR;CHAR;N;37;;
Col3=0;AU_FNAME;AU_FNAME;0;0;20;VARCHAR;CHAR;N;37;;
Col4=0;PHONE;PHONE;0;0;12;CHAR;CHAR;N;37;;
Col5=0;ADDRESS;ADDRESS;0;0;40;VARCHAR;CHAR;N;37;;
Col6=0;CITY;CITY;0;0;20;VARCHAR;CHAR;N;37;;
Col7=0;STATE;STATE;2;0;2;CHAR;CHAR;N;37;;
Col8=0;ZIP;ZIP;0;0;5;CHAR;SHORT;N;37;;
Col9=0;CONTRACT;CONTRACT;0;0;9;BINARY;BINARY;N;37;;
```

Conversion from Host to OLE DB Data Types

The external record description for data files residing on mainframes is configured in a host column description (HCD) file using the SNA OLE DB Management snap-in. This HCD file is used to convert from host EBCDIC data types to ASCII C data types for the PC in the Distributed Data Management (DDM) DLL. These C data types are then mapped to OLE DB data types by the SNAOLEDB DLL.

Default OLE DB Data Types

The following table indicates the default OLE DB data types that result from the mapping of the host data types by the Microsoft OLE DB Provider for AS/400 and VSAM.

Host data type	OLE DB data type	Comments
Binary	DBTYPE_BYTES	A fixed-length array of bytes (unsigned char)
Character	DBTYPE_STR	Null-terminated ASCII character string
Date	DBTYPE_DBDATE	The DBDATE typedef struct defined in OLEDB.H header file.
DBCS	DBTYPE_STR	Null-terminated ASCII character string
DCBS – Mixed Either	DBTYPE_STR	Null-terminated ASCII character string
DCBS – Mixed Open	DBTYPE_STR	Null-terminated ASCII character string
DBCS – Variable	DBTYPE_STR	Null-terminated ASCII character string
DCBS – Variable Mixed Either	DBTYPE_STR	Null-terminated ASCII character string
DCBS – Variable Mixed Open	DBTYPE_STR	Null-terminated ASCII character string
Double	DBTYPE_R8	8-byte floating-point data
Float	DBTYPE_R8	8-byte floating-point data
Long Integer	DBTYPE_I4	4-byte signed integer
Long Variable Binary	DBTYPE_BYTES	A fixed-length array of bytes (unsigned char)
Long Variable Character	DBTYPE_STR	Null-terminated ASCII character string
Packed	DBTYPE_DECIMAL	The DECIMAL structure typedef defined in OLEDB.H. This is an exact numeric value with a fixed precision and fixed scale.
Real	DBTYPE_R4	4-byte floating-point data
Short	DBTYPE_I2	2-byte signed integer
Single	DBTYPE_R4	4-byte floating-point data
Time	DBTYPE_DBTIME	The DBTIME typedef defined in OLEDB.H header file.
Time Stamp	DBTYPE_DBTIMESTAMP	The DBTIMESTAMP typedef defined in OLEDB.H header file.
Variable Binary	DBTYPE_BYTES	A fixed-length array of bytes (unsigned char)
Variable Character	DBTYPE_STR	Null-terminated ASCII character string
Zoned	DBTYPE_NUMERIC	The NUMERIC typedef structure defined in OLEDB.H. This is an exact numeric value with a fixed precision and fixed scale.

The host Binary, VarBinary, and Long VarBinary data types are converted to SQL_C_CHAR type by the DDM DLL and mapped to the DBTYPE_BYTES data type by the SNAOLEDB DLL. The host Zoned data type is converted to SQL_C_CHAR type by the DDM DLL and mapped to the DBTYPE_NUMERIC data type by the SNAOLEDB DLL. The host Packed data type is converted to SQL_C_CHAR type by the DDM DLL and mapped to the DBTYPE_DECIMAL data type by the SNAOLEDB DLL.

DBDATE

The **DBDATE** structure typedef is defined as follows:

```
typedef struct tagDBDATE {
    SHORT  year;
    USHORT month;
    USHORT day
} DBDATE;
```

Members

year
The year (0 to 9999) is measured from 0 A.D.

month
The month ranges from 1 to 12 representing January through December.

day
The day ranges from 1 to a maximum of 31, depending on the number of days in the month.

DBTIME

The **DBTIME** structure typedef is defined as follows:

```
typedef struct tagDBTIME {
    USHORT hour;
    USHORT minute;
    USHORT second
} DBTIME;
```

Members

hour
The hour ranges from 0 to 23.

minute
The minute ranges from 0 to 59.

second
The second ranges from 0 to 59.

DBTIMESTAMP

The **DBTIMESTAMP** structure typedef is defined as follows:

```
typedef struct tagDBTIMESTAMP {
    SHORT  year;
    USHORT month;
    USHORT day;
    USHORT hour;
    USHORT minute;
    USHORT second;
    ULONG  fraction
} DBTIMESTAMP;
```

Members

year
The year (0 to 9999) is measured from 0 A.D.

month
The month ranges from 1 to 12 representing January through December.

day
The day ranges from 1 to a maximum of 31, depending on the number of days in the month.

hour
The hour ranges from 0 to 23.

minute
The minute ranges from 0 to 59.

second
The second ranges from 0 to 59.

fraction
The fraction represents billionths of a second ranging from 0 to 999,999,999.

DECIMAL

The **DECIMAL** typedef structure is an exact numeric value with a fixed precision and fixed scale, stored in the same way as in OLE Automation. The **DECIMAL** typedef structure is defined as follows:

```
typedef struct tagDECIMAL {
    USHORT wReserved;
    union {
        struct {
            BYTE scale;
            BYTE sign;
         };
        USHORT signscale;
    };
    ULONG Hi32;
    union {
        struct {
             ULONG Lo32;
             ULONG Mid32;
          };
         ULONGLONG Lo64;
    };
} DECIMAL;
```

Members

wReserved
This member is reserved and should be 0.

scale
Specifies the number of digits to the right of the decimal point and ranges from 0 to 28.

sign
The sign is 0 if positive, 0x80 if negative.

Hi32
The high part of the integer (32-bit aligned).

Mid32
The middle part of the integer (32-bit aligned).

Lo32
The low part of the integer (32-bit aligned).

For example, to specify the number 12.345, the scale is 3, the sign is 0, and the number stored in the 12-byte integer is 12345.

NUMERIC

The **NUMERIC** typedef structure is an exact numeric value with a fixed precision and fixed scale. The **NUMERIC** typedef structure is defined as follows:

```
typedef struct tagNUMERIC {
    BYTE precision;
    BYTE scale;
    BYTE sign;
    BYTE val[16];
} DB_NUMERIC;
```

Members

precision
The maximum number of digits in base 10.

scale
The number of digits to the right of the decimal point.

sign
The sign is 1 for positive numbers, and 0 for negative numbers.

val
A number stored as a 16-byte scaled integer, with the least-significant byte on the left.

For example, to specify the base 10 number 20.003 with a scale of 4, the number is scaled to an integer of 200030 (20.003 shifted by four tens digits), which is 186AA in hexadecimal. The value stored in the 16-byte integer is 5E 0D 03 00 00 00 00 00 00 00 00 00 00 00 00 00, the precision is the maximum precision, the scale is 4, and the sign is 1.

Character Code Conversions

Character code conversions under the OLE DB Provider for AS/400 and VSAM are controlled by a hierarchy of parameters. When connecting to mainframes all of these parameters are controlled on the PC. However, when connecting to data sources on the AS/400, parameter settings on the AS/400 as well as parameter settings on the PC can be involved.

The Character Code Set Identifier (CCSID) used by the host for a data source can be specified in several locations. The Host CCSID setting used by the OLE DB Provider must bet set to match the CCSID actually used on the host otherwise data loss will occur. Note that some AS/400 systems default to a CCSID of 937 rather than 37 for enabling double-byte character sets (DBCS).

The following table illustrates the separate hierarchy controlling the Host CCSID parameter on the SNA client, AS/400 host, and mainframe host:

SNA client	**AS/400 host**	**Mainframe host**
SNA OLE DB Provider (defaults to U.S./Canada 37)	System (DSPSYSVAL, QCCSID)	None. Determined by the Data Description (HCD file) on the SNA Client.
Data Source (the Host CCSID parameter in the Data Links file describing the Data Source)	User identifier (wrk usrprf)	
Data Description (HCD file)	Job (dspjob, crtjobd)	
	File (dspfd, crtpf)	
	Column (dspffd)	

Only those fields in a Data Source that contain character data will be affected by the Host CCSID parameters and character conversions.

Host CCSID and SNA OLE DB Provider

The Host CCSID setting at the OLE DB provider level defaults to U.S./Canada (37).

Host CCSID and Data Source

The Host CCSID setting at the Data Source level is configured using Data Links for each Data Source. The **Host CCSID** parameter is configured under the **All** tab of the **Data Links** dialog box.

Valid values for the Host CCSID registry setting are any CCSID, including 65535. If the Host CCSID attribute is set to 65535, then no character conversion will occur (the data will be treated as binary). If the Host CCSID setting at the Data Source level does not exist, then the value for Host CCSID defaults to the value at the SNA OLE DB Provider level.

Host CCSID and Data Description

A Host CCSID attribute can be applied at the Data Description level. Each column in a Host Column Description (HCD) file can have a Host CCSID attribute that determines how the character data in the column is to be converted. These attributes in the HCD file at the Data Description level should be configured using the SNA OLE DB Management Console snap-in (see Configuring Data Descriptions). Valid values are any CCSID including 65535.

The Host CCSID attribute at the column Data Description level can be any value and may be empty. A Host CCSID value at the Data Description level overrides the value specified at the Data Source and OLE DB Provider levels. If the Host CCSID is blank, then the value for Host CCSID defaults to the value at the Data Source level. If the Host CCSID attribute is set to 65535, then no character conversion will occur (the data will be treated as binary).

Host CCSID and the Process Binary As Character Parameter

There is a Data Source parameter configurable using Data Links that affects whether binary data is considered as character data and is converted based on the Host CCSID setting. This **Process Binary As Character** parameter defaults to false. If this parameter is false, binary data is not treated as character (binary data is not affected by the Host CCSID setting). If this parameter is set to true, then binary data will be converted based on the Host CCSID setting.

This parameter is configured for each Data Source using Data Links under the **All** tab of the **Data Links** dialog box.

Architecture

The table below lists the architectural components used to support the Microsoft OLE DB Provider for AS/400 and VSAM.

Module	Description
DDMSERV.EXE	The DDM Server implements a source-only DDM Application Requester that communicates with the target DDM Servers running on MVS, OS/390, or OS/400. The DDM Server handles connection management, memory management, file management, and error handling. On Microsoft® Windows 2000 and Windows NT®, DDMSERV runs by default as an auto-started Windows NT service, although this behavior can be changed.
MMCRLIO.DLL	The Data Descriptions Management Console snap-in provided with Host Integration Server and Host Integration Client. This component manages data descriptions and host column description files under the Microsoft® Windows 2000, Windows NT, Windows® 98, and Windows® 95 operating systems.
SNAOLEDB.DLL	The OLE DB provider for AS/400 and VSAM. Server-based and client-based OLE DB and ADO programs call this provider to gain record-level access to host files.

The following additional components must be installed in the system subdirectory below the root directory where Host Integration Server 2000 is installed (setup defaults to C:\Program Files\Host Integration Server) to support the OLE DB Provider for AS/400 and VSAM:

CONV.DLL

DCONV2.DLL

DDM.DLL

DDMAPI.DLL

DDMRLIO.DLL

DDMRC.DLL

DDMSTR.DLL

DDMTCP2.DLL

DDMWAPPC.DLL

SNADDMEL.DLL

SNAOLEDB.MSC

WAPPC32.DLL

WINAPPC.DLL

The following additional components must be installed in the WINDOWS SYSTEM subdirectory (typically WINNT\System32 on Windows 2000 and Windows NT and Windows\System on Windows 98 and Windows 95) to support the OLE DB Provider for AS/400 and VSAM:

MSVBVM50.DLL

MSVBVM60.DLL

The following utilities are also supplied to support the OLE DB Provider for AS/400 and VSAM:

CrtPkg.exe

NewSnaDS.exe

reg2udl.exe

The table below lists the architectural components used to support the Microsoft OLE DB Provider for DB2.

Module	Description

DRDA_VB.EXE	The DRDA Application Requester implemented as a service that communicates with the target DD2 Servers running on MVS, OS/390, or OS/400. The DRDA requester handles connection management, memory management, data access, and error handling. On Microsoft® Windows 2000 and Windows NT®, DRDA_VB runs by default as an auto-started Windows NT service, although this behavior can be changed.
DB2OLEDB.DLL	The OLE DB provider for DB2. Server-based and client-based OLE DB and ADO programs call this provider to gain access to DB2.

The following additional components must be installed in the system subdirectory below the root directory where Host Integration Server 2000 is installed (setup defaults to C:\Program Files\Host Integration Server) to support the OLE DB Provider for DB2:

CONV.DLL

DB2WAPPPC.DLL

DCONV2.DLL

DDMRC.DLL

DDMSTR.DLL

DDMTCP2.DLL

MSEIDRDA.DLL

MSEIDPM.DLL

MSEIDT.DLL

DRDAPERF.DEF

DRDAPERF.INI

SNADDMEL.DLL

WAPPC32.DLL

WINAPPC.DLL

The following additional components must be installed in the WINDOWS SYSTEM subdirectory (typically WINNT\System32 on Windows 2000 and Windows NT and Windows\System on Windows 98 and Windows 95) to support the OLE DB Provider for DB2:

MSVBVM50.DLL

MSVBVM60.DLL

The following utilities are also supplied to support the OLE DB Provider for DB2:

CrtPkg.exe

CrtPkg.dll

NewSnaDS.exe

The table below lists the architectural components used to support the Microsoft ODBC Driver for DB2.

Module	**Description**
DRDA_VB.EXE	The DRDA Application Requester implemented as a service that communicates with the target DD2 Servers running on MVS, OS/390, or OS/400. The DRDA requester handles connection management, memory management, data access, and error handling. On Microsoft® Windows 2000 and Windows NT®, DRDA_VB runs by default as an auto-started Windows NT service, although this behavior can be changed.
MSEIDBC.DLL	The ODBC Driver for DB2 configuration DLL.

MSEIDBD.DLL	The ODBC Driver for DB2.

The following additional components must be installed in the system subdirectory below the root directory where Host Integration Server 2000 is installed (setup defaults to C:\Program Files\Host Integration Server) to support the ODBC Driver for DB2:

CONV.DLL

DB2WAPPPC.DLL

DCONV2.DLL

DDMRC.DLL

DDMSTR.DLL

DDMTCP2.DLL

MSEIDRDA.DLL

MSEIDPM.DLL

MSEIDT.DLL

DRDAPERF.DEF

DRDAPERF.INI

SNADDMEL.DLL

WAPPC32.DLL

WINAPPC.DLL

The following additional components must be installed in the WINDOWS SYSTEM subdirectory (typically WINNT\System32 on Windows 2000 and Windows NT and Windows\System on Windows 98 and Windows 95) to support the ODBC Driver for DB2:

MSVBVM50.DLL

MSVBVM60.DLL

The following utilities are also supplied to support the ODBC Driver for DB2:

CrtPkg.exe

CrtPkg.dll

NewSnaDS.exe

The table below lists the architectural components used to support the Microsoft Host File Transfer ActiveX Control.

Module	**Description**
DDMSERV.EXE	The DDM Server implements a source-only DDM Application Requester that communicates with the target DDM Servers running on MVS, OS/390, or OS/400. The DDM Server handles connection management, memory management, file management, and error handling. On Microsoft® Windows 2000 and Windows NT®, DDMSERV runs by default as an auto-started Windows NT service, although this behavior can be changed.
MMCRLIO.DLL	The Data Descriptions Management Console snap-in provided with Host Integration Server and Host Integration Client. This component manages data descriptions and host column description files under the Microsoft® Windows 2000, Windows NT, Windows® 98, and Windows® 95 operating systems.
MSEIGFT.DLL	The Host File Transfer ActiveX Control library.

The following additional components must be installed in the system subdirectory below the root directory where Host Integration Server 2000 is installed (setup defaults to C:\Program Files\Host Integration Server) to support the Host File Transfer ActiveX Control:

CONV.DLL

DCONV2.DLL

DDM.DLL

DDMAPI.DLL

DDMRLIO.DLL

DDMRC.DLL

DDMSTR.DLL

DDMTCP2.DLL

DDMWAPPC.DLL

WAPPC32.DLL

WINAPPC.DLL

WINCSV.DLL

WINCSV32.DLL

The following additional components must be installed in the WINDOWS SYSTEM subdirectory (typically WINNT\System32 on Windows 2000 and Windows NT and Windows\System on Windows 98 and Windows 95) to support the Host File Transfer ActiveX Control:

MSVBVM50.DLL

MSVBVM60.DLL

SDK Components for Data Integration

The Microsoft® Host Integration Server 2000 SDK contains software components used for data integration applications and data access. The components used for data access are described in the following topics.

Program and DLL Files for Data Integration

The following executable system file and DLL library files are included with the Host Integration Server 2000 SDK for use with the OLE DB Provider for AS/400 and VSAM, the OLE DB Provider for DB2, the ODBC Driver for DB2, the AS/400 Data Queue ActiveX Control, and the Host File Transfer ActiveX Control:

File name	Description
conv.dll	DDM support library used for ASCII-to-EBCDIC conversion during host logon.
CrtPkg.dll	Support library for DB2 create package utility.
CrtPkg.exe	Create package utility for use with the OLE DB Provider for DB2 and the ODBC Driver for DB2.
db2oledb.dll	The OLE DB provider for DB2.
dconv2.dll	DDM support library used for data conversion.
ddm.dll	DDM Record Level I/O Requester library that communicates with the DDM Server.
ddmapi.dll	DDM Record Access Library.
DDMRC.dll	DDM support library containing resource strings used by error-handling routines.
ddmrlio.dll	DDM support library used by snaoledb.dll.
ddmserv.exe	The DDM Server handles connection management, memory management, file management, and error handling.
DDMStr.dll	DDM support library containing the OLE DB provider error string resources.
ddmtcp2.dll	DDM support library used by TCP/IP by the OLE DB Provider for DB2 and the ODBC Driver for DB2.
ddmwappc.dll	DDM support library used for OLE DB Data configuration and communication with the WINAPPC.DLL.
drda_vb.exe	The DRDA Server handles connection management, memory management, file management, and error handling.
mmcrlio.dll	The Data Descriptions Management Console snap-in support library used to create or modify data descriptions for use with the OLE DB Provider for AS/400 and VSAM or with the Host File Transfer ActiveX Control.
mmcrlio.chm	Help file for the MMC snap-in for the OLE DB Provider for AS/400 and VSAM.
mmcRLIO.msc	The Data Descriptions Management Console snap-in utility used to configure or modify data description files for use with the OLE DB Provider for AS/400 and VSAM or with the Host File Transfer ActiveX Control.

msei db2c .dll	The ODBC Driver for DB2 configuration DLL (installed in the Windows 2000 or Windows NT System 32 directory).
msei db2d .dll	The ODBC Driver for DB2 (installed in the Windows 2000 or Windows NT System 32 directory).
msei drda. dll	Support library for use by the OLE DB Provider for DB2 and the ODBC Driver for DB2 used to provide the DRDA agent for t he DB2 provider.
MSEI DPM .dll	Support library for use by the OLE DB Provider for DB2 and the ODBC Driver for DB2 used for DB2 performance monitorin g.
MSEI DT.dl l	Support library for use by the OLE DB Provider for DB2 and the ODBC Driver for DB2 used for DB2 data transformation.
MSEI GFT. dll	The Host File Transfer ActiveX Control library.
MSEI GDQ .dll	The AS/400 Data Queues ActiveX Control library.
MSV BVM 50.D LL	Visual Basic 5.0 runtime dynamic link library.
Msv bvm 60.dl l	Visual Basic 6.0 runtime dynamic link library.
New Sna DS.e xe	A utility program to create new Data Links UDL files used with Host Integration Server 2000.
reg2 udl.e xe	A utility program to convert Win32 registry-based data sources used with previous versions of the OLE DB Provider for AS/ 400 and VSAM supplied with SNA Server 4.0 to Data Links UDL files used with Host Integration Server 2000. This file is loca ted in the Options\Maintenance folder on the Host Integration Server CD-ROM and is not installed as part of the product.
snad dmel .dll	Event log resource file.
snao ledb. dll	The OLE DB provider for AS/400 and VSAM.

The following executable system file and DLL library files were included with SNA Server 4.0 Service Pack 3 and Service Pack 4 for use with the OLE DB Provider for AS/400 and VSAM, the OLE DB Provider for DB2, and the ODBC Driver for DB2:

File na me	**Description**
CONV.D LL	DDM support library used for ASCII-to-EBCDIC conversion during host logon.
DB2OLE DB.DLL	The OLE DB provider for DB2.
DB2SER V.EXE	The DB2 Server handles connection management, memory management, file management, and error handling.
DB2WA PPC.DLL	Support library used for OLE DB Data configuration and communication with the WINAPPC.DLL.
DCONV. DLL	DDM support library used for data conversion.
DDM.DL L	DDM Record Level I/O Requester library that communicates with the DDM Server.

DDMAPI.DLL	DDM Record Access Library.
DDMRLIO.DLL	DDM support library used by SNAOLEDB.DLL.
DDMSERV.EXE	The DDM Server handles connection management, memory management, file management, and error handling.
DDMSQL.DLL	DDM support library used by DB2OLEDB.DLL.
DDMSTR.DLL	DDM support library containing the OLE DB provider error string resources.
DDMTCP2.DLL	DDM support library used by TCP/IP by the OLE DB Provider for DB2 and the ODBC Driver for DB2. (DDMTCP.DLL was supplied with the earlier SNA Server 4.0.)
DDMTCP.DLL	DDM support library used by TCP/IP by the OLE DB Provider for AS/400 and VSAM.
DDMWAPPC.DLL	DDM support library used for OLE DB Data configuration and communication with the WINAPPC.DLL.
DRDA.DLL	DRDA support library.
DRDAAPI.DLL	DRDA support library.
DRDAPERF.DEF	Performance Monitor support file.
DRDAPERF.INI	Performance Monitor support file.
MMCRLIO.DLL	The Data Descriptions OLE DB Management Console snap-in used to create or modify data descriptions for use with the OLE DB Provider for AS/400 and VSAM.
MMCRLIO.HLP	Help file for the MMC snap-in for the OLE DB Provider for AS/400 and VSAM.
MSDB2BAS.DLL	Support library for ODBC Driver for DB2.
MSDB2OLE.DLL	Support library for ODBC Driver for DB2.
MSDB2OLE.HLP	Help file for the ODBC Driver for DB2.
MSDB2UTL.DLL	Utility support library for ODBC Driver for DB2.
MSVBVM50.DLL	Visual Basic 5.0 runtime dynamic link library.
PMHLPRDR.DLL	Performance Monitor support file.
PMPRXYDR.DLL	Performance Monitor support file.
REG2UDL.EXE	A utility program to convert Win32 registry-based data sources used with previous versions of the OLE DB Provider for AS/400 and VSAM supplied with SNA Server 4.0 to Data Links UDL files used with Host Integration Server 2000.
SNADDMEL.DLL	Event log resource file.
SNAOLEDB.DLL	The OLE DB provider for AS/400 and VSAM.
SNAOLEDB.MSC	SNA OLE DB Management Console snap-in utility used to configure data sources in the registry and host column description files.

Symbol Files for Data Integration

The following symbol files for use when debugging are included with Host Integration Server 2000 for use with the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, the Microsoft® ODBC Driver for DB2, the Microsoft Host File Transfer ActiveX Control, and the Microsoft Data Queues ActiveX Control. These files are installed as part of the Host Integration Server package and a copy of these files are also located on the Host Integration Server CD-ROM under the Support\Symbols folder:

File name	Description
DLL\conv.dbg	Symbols from Conv.dll
DLL\CrtPkg.dbg	Symbols from CrtPkg.dll
EXE\CrtPkg.dbg	Symbols from CrtPkg.exe
DLL\db2oledb.dbg	Symbols from db2oledb.dll
DLL\dconv2.dbg	Symbols from dconv2.dll
DLL\ddm.dbg	Symbols from ddm.dll
DLL\ddmapi.dbg	Symbols from ddmapi.dll
DLL\ddmrlio.dbg	Symbols from ddmrlio.dll
DLL\ddmserv.dbg	Symbols from ddmserv.exe
DLL\DDMStr.dbg	Symbols from DDMStr.dll
DLL\ddmtcp2.dbg	Symbols from ddmtcp2.dll
DLL\ddmwappc.dbg	Symbols from ddmwappc.dll
EXE\DRDA_VB.dbg	Symbols from drdda_vb.exe
DLL\mmcrlio.dbg	Symbols from mmcrlio.dll
DLL\mdeidb2c.dbg	Symbols from MSEIDB2C.DLL
DLL\mseidb2d.dbg	Symbols from MSEIDB2D.DLL
DLL\mseidrda.dbg	Symbols from mseidrda.dll
DLL\MSEIDPM.dbg	Symbols from MSEIDPM.dll
DLL\MSEIDSBD.dbg	Symbols from MSEIDSBD.DLL
DLL\MSEIDSBN.dbg	Symbols from MSEIDSBN.DLL
DLL\MSEIDSBP.dbg	Symbols from MSEIDSBP.DLL
DLL\MSEIDT.dbg	Symbols from MSEIDT.dll
DLL\MSEIGFT.dbg	Symbols from MSEIGFT.dll
DLL\MSEIGDQ.dbg	Symbols from MSEIGDQ.dll
EXE\NewSnaDS.dbg	Symbols from NewSnaDS.exe
DLL\reg2udl.dbg	Symbols from REG2UDL.EXE
DLL\snaddmel.dbg	Symbols from snaddmel.dll
DLL\snaoledb.dbg	Symbols from snaoledb.dll

The following symbol files for use when debugging were included with SNA Server 4.0 Service Pack 3 and Service Pack 4 for use with the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, and the Microsoft® ODBC Driver for DB2:

File name	Description
CONV.DBG	Symbols from CONV.DLL
CRTPKG.DBG	Symbols from CRTPKG.EXE
CRTPKGW.DBG	Symbols from CRTPKGW.EXE
DB2OLEDB.DBG	Symbols from DB2OLEDB.DLL
DB2SERV.DBG	Symbols from DB2SERV.EXE
DB2WAPPC.DBG	Symbols from DBWAPPC.DLL
DCONV.DBG	Symbols from DCONV.DLL
DDM.DBG	Symbols from DDM.DLL
DDMAPI.DBG	Symbols from DDMAPI.DLL
DDMRLIO.DBG	Symbols from DDMRLIO.DLL
DDMSERV.DBG	Symbols from DDMSERV.EXE
DDMSQL.DBG	Symbols from DDMSQL.DLL
DDMSTR.DBG	Symbols from DDMSTR.DLL

DDMTCP.DBG	Symbols from DDMTCP.DLL
DDMTCP2.DBG	Symbols from DDMTCP2.DLL
DDMWAPPC.DBG	Symbols from DDMAPPC.DLL
DRDA.DBG	Symbols from DRDA.DLL
DRDAAPI.DBG	Symbols from DRDAAPI.DLL
MMCRLIO.DBG	Symbols from MMCRLIO.DLL
MSDB2BAS.DBG	Symbols from MSDB2BAS.DLL
MSDB2OLE.DBG	Symbols from MSDB2OLE.DLL
MSDB2UTL.DBG	Symbols from MSDB2UTL.DLL
PMHLPRDR.DBG	Symbols from PMHLPRDR.DLL
PMPRXYDR.DBG	Symbols from PMPRXYDR.DLL
REG2UDL.DBG	Symbols from REG2UDL.EXE
SNADDMEL.DBG	Symbols from SNADDMEL.DLL
SNAOLEDB.DBG	Symbols from SNAOLED.DLL

Header Files for Data Integration

Provider-specific header files needed to build the ADO, OLE DB, and ODBC sample applications are included with the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, and the Microsoft® ODBC Driver for DB2. These header files are located in the SDK\Include subdirectory on the Host Integration Server 2000 CD-ROM.

The following provider-specific files are provided for developing applications using ADO, OLE DB, and ODBC using the OLE DB Provider for AS/400 and VSAM, the OLE DB Provider for DB2, and the ODBC Driver for DB2:

File name	Description
db2oledb.h	GUID definitions, enumeration constants, and error codes for use with the OLE DB Provider for DB2.
snaoledb.h	GUID definitions, enumeration constants, and error codes for use with the OLE DB Provider for AS/400 and VSAM.

The following header files are provided as part of the OLE DB version 2.1 and ADO version 2.1 SDK. These OLE DB and ADO header files are generally required to build applications in C or C++ using the Host Integration Server SDK and may be downloaded from the Microsoft Universal Data Access Free Downloads page. http://go.microsoft.com/fwlink/?LinkId=12754 as part of the Microsoft Data Access SDK 2.1.

File name	Description
ADOID.H	ADO GUID definitions.
ADOINT.H	ADO Interface header for calling from C++.
ADOJAVAS.INC	ADO constants include file for JavaScript.
ADOVBS.INC	ADO constants include file for VBScript.
MSDADC.H	OLE DB Data Conversion include file.
MSDAGUID.H	Microsoft Data Access GUID definitions.
MSDASC.H	Microsoft Data Access Datalinks interfaces.
MSDASQL.H	OLE DB Provider for ODBC data.
ODBCINST.H	ODBC install include file.
OLEDB.H	Primary OLE DB include file.
OLEDBERR.H	OLE DB error include file.
OLEDBJVS.INC	OLE DB error constant include file for JavaScript.
OLEDBVBS.INC	OLE DB error constant include file for VBScript.
SQL.H	Main include file for ODBC core functions.
SQLEXT.H	Include file for applications using Microsoft SQL extensions.
SQLTYPES.H	Defines types used in ODBC.
SQLUCODE.H	Unicode include file for ODBC core functions.
TRANSACT.H	Transactions header file.

These include files are also included in the SDK\INC subdirectory on the SNA Server 4.0 Service Pack 3 and Service Pack 4 CD-ROM for Win32 platforms on I386 and Alpha.

Import Library Files for OLE DB Data Integration

The following import library files are on Microsoft Developer Network CD-ROMs and as a downloadable Software Development Kit from the the Microsoft Universal Data Access Free Downloads Web site at http://go.microsoft.com/fwlink/?LinkId=12754 provided as part of the Microsoft Data Access SDK 2.0 and the Microsoft Data Access SDK 2.1 update:

File name	Description
ADOID.LIB	GUID library for client C++ classes
MSDASC.LIB	Data Links library
OLEDB.LIB	The OLE DB library
OLEDBD.LIB	The OLE DB debug library

The MSDASC.LIB, OLEDB.LIB, and OLEDBD.LIB library files are also included in the SDK\LIB\WINNT subdirectory on the SNA Server 4.0 Service Pack 3 and Service Pack 4 CD-ROM for Win32 platforms on I386 and Alpha.

Import Library Files for ODBC Data Integration

The following import library files are on Microsoft Developer Network CD-ROMs and as a downloadable Software Development Kit from the the Microsoft Universal Data Access Free Downloads Web site at http://go.microsoft.com/fwlink/?LinkId=12754 provided as part of the Microsoft Data Access SDK 2.1 update:

File name	Description
GTRTST32.LIB	An ODBC support library
ODBC32.LIB	The main ODBC library
ODBCCP32.LIB	An ODBC support library

These files are also included with the Microsoft® ODBC Driver for DB2 in the SDK\LIB\WINNT subdirectory on SNA Server 4.0 Service Pack 3 and Service Pack 4 CD-ROM for Win32 platforms on I386 and Alpha.

OLE DB and ADO Import Library Files

The following import library files are on Microsoft Developer Network CD-ROMs and as a downloadable Software Development Kit from the Microsoft Universal Data Access Free Downloads Web site at http://go.microsoft.com/fwlink/?LinkId=12754 provided as part of the Microsoft Data Access SDK 2.0 and the Microsoft Data Access SDK 2.1 update:

File name	Description
ADOID.LIB	GUID library for client C++ classes
MSDASC.LIB	Data Links library
OLEDB.LIB	The OLE DB library
OLEDBD.LIB	The OLE DB debug library

The MSDASC.LIB, OLEDB.LIB, and OLEDBD.LIB library files are also included in the SDK\LIB\WINNT subdirectory on the SNA Server 4.0 Service Pack 3 and Service Pack 4 CD-ROM for Win32 platforms on I386 and Alpha.

Data Integration Samples

This section of the Microsoft® Host Integration Server 2000 Developer's Guide describes the sample applications that implement data integration using DB providers, drivers, and ActiveX® controls.

This section contains:

Sample Programs for Data Access

The source code and executable files for several sample programs that illustrate using the Microsoft® OLE DB Provider for AS/400 and VSAM, the Microsoft® OLE DB Provider for DB2, and the Microsoft® ODBC Driver for DB2 are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\DataAccess subdirectory on the Host Integration Server 2000 CD-ROM. These files are also copied to your hard drive during Host Integration Server and Host Integration Client installation when the SDK option is selected.

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\DataAccess subdirectory below where the MSDN Platform SDK has been installed.

These sample programs include the files in the following subdirectories:

Subdirectory	Description
DRDA_ASP	ADO sample script written in Microsoft® Active Server Pages (ASP) using the OLE DB Provider for DB2
DRDA_VB	ADO sample program written in Microsoft® Visual Basic® using the OLE DB Provider for DB2
DRDA_VBS	ADO sample script written in Microsoft® VBScript using the OLE DB Provider for DB2
RLIO_ASP	ADO sample script written in Microsoft® Active Server Pages (ASP) using the OLE DB Provider for AS/400 and VSAM
RLIO_VB	ADO sample program written in Microsoft® Visual Basic® using the OLE DB Provider for AS/400 and VSAM
RLIO_VBS	ADO sample script written in Microsoft® VBScript using the OLE DB Provider for AS/400 and VSAM

The file named SAMPLE.HCD located in the DataAccess subdirectory contains two examples of a Host Column Description (HCD) file required to access host VSAM files using the OLE DB Provider for AS/400 and VSAM. Please see documentation on the Host Column Description in the Data Access Reference section for more information. The SAMPLE.HCD file is described in this same reference section as well.

Several sample programs with source code are provided with Host Integration Server 2000 that illustrate data access. These sample programs include the following:

- Visual Basic ADO sample program This application allows you to open a connection through the OLE DB Provider, open a recordset, and browse the records of a host file. Two versions of this sample application are provided. The sample in the DRDA_VB folder is for use with the OLE DB Provider for DB2. The sample in the RLIO_VB folder is for use with the OLE DB Provider for AS/400 and VSAM. Seeking on a key value in a host indexed file is also possible with this sample when the version of this sample is used with the OLE DB Provider for AS/400 and VSAM.
- Visual Basic Script ADO sample program This VBScript application allows you to open a connection through the OLE DB Provider, open a recordset, and browse the records of a host file. Two versions of this sample application are provided. The sample in the DRDA_VBS folder is for use with the OLE DB Provider for DB2. The sample in the RLIO_VBS folder is for use with the OLE DB Provider for AS/400 and VSAM.
- Active Server Pages (ASP) sample programs Two ASP sample programs are included. One is a report application that allows you to return and display the first ten records in a host file to a client browser in a table view. The second sample allows you to display columns of a host file in a form view. Two versions of these sample applications are provided. The samples in the DRDA_ASP folder are for use with the OLE DB Provider for DB2. The samples in the RLIO_ASP folder are for use with the OLE DB Provider for AS/400 and VSAM.

Several additional SDK sample files for data access are included with Microsoft Data Access Components (MDAC) SDK and are available as part of the Microsoft MSDN Platform SDK. These data access sample programs include the following files:

Sample Program	Description
ODBCQuery	The ODBC Query demo application written in C. This sample can be used with the ODBC Driver for DB2.
RowsetViewer	The OLE DB RowsetViewer sample application written in C++. This sample can be with either the OLE DB Provider for DB2 or the OLE DB Provider for AS/400 and VSAM.

These sample programs with source code are provided as part of the MDAC SDK and the MSDN Platform SDK can be used with the data access features of Host Integration Server 2000:

- OLE DB RowsetViewer sample program This application written on C++ allows you to connect to the OLE DB Provider, open a table window, type the host file name or database, return a rowset, and browse the contents. You can navigate the table,

using seek and set range on indexed files when the version of this sample is used with the OLE DB Provider for AS/400 and VSAM.

Before running any of these examples with the OLE DB Provider for AS/400 and VSAM, it is necessary to create a data file on the SNA host that will be used for demonstration purposes. After a suitable file is created on the host, it is necessary to run the Data Descriptions management console snap-in application in order to configure the appropriate default settings for the OLE DB Provider for AS/400 and VSAM, such as APPC mode, the character code set Identifier (CCSID) to be used when converting host data, and the Code Page to be used on the local PC.

If the data is on a mainframe host, it is also necessary to configure a data description (create a host column description file) using the Data Descriptions management console snap-in. The host column description specifies the column description of the data file on the host and how the OLE DB Provider for AS/400 and VSAM should convert data from the host to the PC. Although an HCD file is unnecessary to access data files on the AS/400, a host column description can be specified that will override the default conversions.

The following sections discuss setting up and using each sample application in more detail.

Visual Basic ADO Sample

The SDK\Samples\DataAccess folder contains an ADO sample application written in Visual Basic version 6.0 that accesses data provided by an OLE DB Provider. Two versions of this sample are supplied. One sample is for use with the Microsoft OLE DB Provider for AS/400 and VSAM (source files located in the RLIO_VB subdirectory). One sample is for use with the Microsoft OLE DB Provider for DB2(source files located in the DRDA_VB subdirectory).

This sample illustrates using ADO to access data on a host.

In order to use this sample, the ConnectionString property will need to be changed to point to a host you can access and a data file or DB2 database to browse on the host.

This application allows you to open a connection through the OLE DB Provider, open a recordset, and browse the records on a host file. This sample is built with ActiveX® Data Objects version 2.0. Before building this Visual Basic sample, be sure to include the Microsoft ActiveX Data Objects 2.0 Library in your project. You can do this from inside the Visual Basic Integrated Development Environment by selecting **References** on the **Project** menu.

The sample application can be built with Microsoft Visual Basic version 5.0 or later.

Visual Basic Script ADO Sample

The SDK\Samples\DataAccess folder contains a sample web client application written in VBScript using ADO designed to access data provided by an OLE DB provider. Two versions of this sample are supplied. One sample is for use with the Microsoft® OLE DB Provider for AS/400 and VSAM (source files located in the RLIO_VBS subdirectory). One sample is for use with the Microsoft® OLE DB Provider for DB2 (source files located in the DRDA_VBS subdirectory).

This web client application allows you to open a connection through the OLE DB Provider, open a recordset, and browse the records of a host file.

In order to use this sample, the ConnectionString property will need to be changed to point to a host you can access and a data file or DB2 database to browse on the host.

Active Server Pages Samples

The SDK\Samples\DataAccess folder contains an Active Server Pages sample web server application designed to access data provided by an OLE DB provider. Two versions of this sample application are supplied. One sample is for use with the Microsoft® OLE DB Provider for AS/400 and VSAM (source files located in the RLIO_ASP subdirectory). One sample is for use with the Microsoft® OLE DB Provider for DB2 (source files located in the DRDA_ASP subdirectory).

These sample applications require Microsoft Internet Information Server version 3.0 or higher with Active Server Pages installed.

REPORT.ASP will return and display the first ten (10) records in a host file to a client browser in a table view.

OLEDBDDM.ASP and GLOBAL.ASA combine to display columns of a host file in a form view.

You will need to edit the host table names or DB2 database names in order to use these samples in your environment.

OLE DB RowsetViewer Sample

The Microsoft® Data Access SDK and the Microsoft Developer Network (MSDN) Platform SDK contain an OLE DB RowsetViewer sample application written in C++. This application allows you to connect to either the Microsoft OLE DB Provider for AS/400 and VSAM or the Microsoft OLE DB Provider for DB2, open a table window, type the host file name or DB2 database, return a rowset, and browse the contents. You can navigate the table, using seek and set range on indexed files when this sample is used with the OLE DB Provider for AS/400 and VSAM.

To open a rowset using the OLE DB Provider for AS/400 and VSAM, type the following command in the command window:

```
exec open library/filename
```

You can use "/" or "." depending on whether you are accessing a host mainframe or an AS/400 file. Note that if a "." appears in the host filename, then the filename must be enclosed in double quotes.

This sample is written to access bookmarkable AS/400 and VSAM file types only. You will not be able to access mainframe SAM, ESDS, PDS/PDSE because these data set types do not support bookmarks.

You may not be able to build this sample application, if certain OLE DB header and library files are not installed. The Microsoft® Host Integration Server 2000 CD-ROM does not include a copy of the Microsoft Data Access SDK (DASDK) or the earlier OLE DB SDK. If you do not have the Data Access SDK or the OLE DB SDK installed, then you need to copy the following header files from the Microsoft Platform SDK into your project.

```
adoid.h
adoint.h
db2oledb.h
msdadc.h
msdaguid.h
msdasql.h
oledb.h
oledberr.h
snaoledb.h
transact.h
```

If you do not have the Data Access SDK or the Platform SDK installed, then you will need to build your application using one of the following library files copied from the Microsoft Platform SDK.

```
OLEDB.LIB
OLEDBD.LIB (debug version of OLEDB library)
```

Note that only i386 LIB files are supported using Host Integration Server 2000. On the Microsoft® SNA Server 4.0 CD-ROM, different OLE DB LIB files are supplied for i386 and Alpha processors.

Sample Programs for Data Queues

The source code for several sample programs that illustrate using the Microsoft® Data Queue ActiveX Control are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\DataQueues subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\DataQueues subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\DataQueues subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files in the following subdirectories:

Subdirectory	**Description**
DQChatC	A chat program written in Microsoft® Visual C++ that illustrates using the Data Queue ActiveX Control and AS/400 data queues.
DQChatB	A chat program written in Microsoft Visual Basic that illustrates using the Data Queue ActiveX Control and AS/400 data queues.

The DQChatC sample is designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio 6.0 or Microsoft® Visual Studio .NET interactive development environment (IDE). To build the DQChatC sample using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory(by default, C:\Program Files\Microsoft Visual Studio\VC98\Bin for VS6 or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools for VS.NET))

To build the DQChatC sample, open an MS-DOS Command Prompt window, navigate to DataQueues\DQChatC subdirectory, and invoke NMAKE.

To build the DQChatC using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (DataQueues\DQChatC\dqchatc.vcproj) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

DQChatC Sample

The sample program DQChatC is a Microsoft Visual C++ program that demonstrates the use of Data Queues in the form of an Internet "chat room". Data, in the form of messages or "chat" strings, is transferred back and forth through a data queue. The DQChatC program was created using the Visual C++ AppWizard.

The files initially created by AppWizard included for this sample program are:

- DQChatC.dsp

  The Visual C++ project file used to build this sample. Other users can share the project (.dsp) file, but they should export any makefiles locally.

- DQChatC.h

  This is the main header file for the application. It includes other project specific headers (including Resource.h) and declares the CDQChatCApp application class.

- DQChatC.cpp

  This is the main application source file that contains the application class CDQChatCApp.

- DQChatC.rc

  This is a listing of all of the Microsoft Windows resources that the program uses. It includes the icons, bitmaps, and cursors that are stored in the RES subdirectory. This file can be directly edited in Microsoft Visual C++.

- DQChatC.clw

  This file contains information used by the Visual C++ ClassWizard to edit existing classes or add new classes. ClassWizard also uses this file to store information needed to create and edit message maps and dialog data maps and to create prototype member functions.

- DQChatCDlg.h, DQChatCDlg.cpp - the dialog

  These files contain your CDQChatCDlg class. This class defines the behavior of your application's main dialog. The dialog's template is in DQChatC.rc, which can be edited in Microsoft Visual C++.

- eigdataqueue.cpp

  This source file contains machine generated IDispatch wrapper classes created by Microsoft Visual C++.

- eigdataqueue.h

  This header file contains machine generated IDispatch wrapper classes created by Microsoft Visual C++.

- StdAfx.h, StdAfx.cpp

  These files are used to build a precompiled header (PCH) file named DQChatC.pch and a precompiled types file named StdAfx.obj used by Visual C++.

- Resource.h

  This is the standard header file, which defines new resource IDs. Microsoft Visual C++ reads and updates this file.

The following files are located in the DataQueues\DQChatC\res subdirectory:

- DQChatC.ico

  This is an icon file, which is used as the application's icon. This icon is included by the main resource file DQChatC.rc.

- res\DQChatC.rc2

  This file contains resources that are not edited by Microsoft Visual C++. All resources that are not able to be edited should be placed in this file.

Sample Programs for Host File Transfer

The source code for several sample programs that illustrate using the Microsoft® Host File Transfer ActiveX Control are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\FileTransfer subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\FileTransfer subdirectory below where the Host Integration Server software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\FileTransfer subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files in the following subdirectories:

Subdirectory	Description
TestConnectC	A sample script written in Microsoft® Visual C++ using the Host File Transfer ActiveX Control.
TestConnectVB	A sample program written in Microsoft® Visual Basic® using the Host File Transfer ActiveX Control.

The TestConnectC sample is designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio 6.0 or Microsoft® Visual Studio .NET interactive development environment (IDE). To build the TestConnectC sample using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory(by default, C:\Program Files\Microsoft Visual Studio\VC98\Bin for VS6 or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools for VS.NET))

To build the TestConnectC sample, open an MS-DOS Command Prompt window, navigate to FileTransfer\TestConnectC subdirectory, and invoke NMAKE.

To build the TestConnectC using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (FileTransfer\TestConnectC\testconnectc.vcproj) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

SNA Application Programming

This section describes how to create applications in a Systems Network Architecture (SNA) environment.

This section contains:

- APPC Applications
- CPI-C Applications
- LUA Applications
- 3270 Emulator Interface Specifications
- AFTP File Transfer Protocol

APPC Applications

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information required to develop C-language applications that use Advanced Program-to-Program Communications (APPC) to exchange data in a Systems Network Architecture (SNA) environment.

This section contains:

About the APPC Guide

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop C-language applications that use Advanced Program-to-Program Communications (APPC) to exchange data in a Systems Network Architecture (SNA) environment.

This guide is intended for the programmer writing applications that use CPI-C to exchange data. It provides conceptual information and detailed reference information.

To use this guide effectively, you should be familiar with:

- Microsoft® Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows® 2000
    - Microsoft Windows NT®
    - Microsoft Windows 98
    - Microsoft Windows 95
- SNA concepts

Operating Systems Support for APPC Development

This section of the guide contains information relating to following operating systems:

- Microsoft® Windows® 2000
- Microsoft Windows NT®
- Microsoft Windows 98
- Microsoft Windows 95
- Microsoft Windows version 3.*x*
- Microsoft MS-DOS®
- OS/2

Microsoft Host Integration Server 2000 supports the development of APPC applications for Windows 2000, Windows NT, Windows 98, and Windows 95. Under these operating systems, support for APPC applications is provided only for the Win32® subsystem.

The previous Microsoft SNA Server product also supported the development of APPC applications for Windows 3.*x*, MS-DOS, and OS/2. Most APPC applications developed for Windows 3.x, MS-DOS, and OS/2 with SNA Server can be used with Host Integration Server 2000. The Windows 3.x, MS-DOS, and OS/2 interface is described here for completeness, but Windows 3.x, MS-DOS, or OS/2 APPC application development is not supported using Host Integration Server.

Finding Further Information

This guide does not describe the products, architectures, or standards developed by other companies or organizations. For information about the Microsoft® Windows® graphical environment, Windows NT®, Windows 95, Windows 98, and the MS-DOS® and IBM OS/2 operating systems, consult your product documentation.

The following documents provide additional information about SNA Server APIs:

- *Microsoft Host Integration Server CPI-C Programmer's Guide*
- *Microsoft Host Integration LUA Programmer's Guide*

For more information about SNA and about 3270 information display systems, see the following manuals:

- *IBM 3270 Information Display System: 3274 Control Unit Description and Programmer's Guide*
- *IBM 3270 Information Display System: Color and Programmed Symbols*
- *IBM 3270 Information Display System: 3274 Control Unit Display Station: Operator's Guide*
- *IBM Systems Network Architecture: Technical Overview*
- *IBM Systems Network Architecture: Concepts and Products*
- *IBM Advanced Communications Function Products Installation Guide*
- *IBM Installation and Resource Definition*
- *IBM 9370 LAN Token Ring Support*
- *IBM SNA Format and Protocol Reference Manual: Architectural Logic*

For background information about logical unit (LU) 6.2, APPC, or the Common Programming Interface for Communications (CPI-C), see the following manuals:

- *IBM Systems Network Architecture: Introduction to APPC*
- *IBM Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2*
- *IBM SNA: Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM SNA: Formats*
- *IBM SNA: Technical Overview*
- *IBM SNA: ACF/VTAM Programming for LU Type 6.2*

APPC Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about programming the Advanced Program-to-Program Communications (APPC) in a distributed processing environment.

This section contains:

Introduction to APPC

This section introduces the fundamental concepts of Advanced Program-to-Program Communications (APPC) in a distributed processing environment. These concepts include:

- APPC verbs
- Microsoft® Windows® APPC extensions
- Using APPC verbs in C programs
- Operating system considerations

Detailed descriptions of APPC verbs are provided in APPC Management Verbs, APPC TP Verbs, and APPC Conversation Verbs.

APPC is an application programming interface (API) that allows peer-to-peer communications in a Systems Network Architecture (SNA) environment. Through APPC, programs distributed across a network can work together, communicating with each other and exchanging data, to accomplish a single processing task such as querying a remote database, copying a remote file, or sending and receiving electronic mail.

APPC Verb Overview

APPC verbs fall into three categories: management, transaction program (TP), and conversation.

Management Verbs

Management verbs provide management functions. They are:

ACTIVATE_SESSION

CNOS

DEACTIVATE_SESSION

DISPLAY

TP Verbs

TP verbs start and end TPs, and get and set TP properties. They are:

GET_TP_PROPERTIES

SET_TP_PROPERTIES

TP_ENDED

TP_STARTED

Conversation Verbs

Conversation verbs enable TPs to allocate and deallocate conversations, send and receive data, and change conversation states. The conversation verbs are listed in the following table.

Conversation verbs fall into two groups: mapped conversation verbs and basic conversation verbs. The mapped conversation is intended for programs that use the conversation directly. The basic conversation is intended for more complex programs that provide services to other users. In typical situations, end-user TPs use mapped conversations and service TPs use basic conversations.

Mapped conversation verbs can only be issued by a TP in mapped conversations, while basic conversation verbs are reserved for basic conversations. There is one exception to this rule: **ALLOCATE** can be used to start either a basic or a mapped conversation.

Mapped conversation verbs	Basic conversation verbs
MC_ALLOCATE	**ALLOCATE**
MC_CONFIRM	**CONFIRM**

MC_CONFIRMED	**CONFIRMED**
MC_DEALLOCATE	**DEALLOCATE**
MC_FLUSH	**FLUSH**
MC_GET_ATTRIBUTES	**GET_ATTRIBUTES**
GET_LU_STATUS	**GET_LU_STATUS**
GET_STATE	**GET_STATE**
GET_TYPE	**GET_TYPE**
MC_POST_ON_RECEIPT	**POST_ON_RECEIPT**
MC_PREPARE_TO_RECEIVE	**PREPARE_TO_RECEIVE**
RECEIVE_ALLOCATE	**RECEIVE_ALLOCATE**
MC_RECEIVE_AND_POST	**RECEIVE_AND_POST**
MC_RECEIVE_AND_WAIT	**RECEIVE_AND_WAIT**
MC_RECEIVE_IMMEDIATE	**RECEIVE_IMMEDIATE**
MC_RECEIVE_LOG_DATA	**RECEIVE_LOG_DATA**
MC_REQUEST_TO_SEND	**REQUEST_TO_SEND**
MC_SEND_CONVERSATION	**SEND_CONVERSATION**
MC_SEND_DATA	**SEND_DATA**
MC_SEND_ERROR	**SEND_ERROR**
MC_TEST_RTS	**TEST_RTS**

Mapped and basic verbs have the same functions in their respective types of conversation. For example, **MC_CONFIRM** performs the same function in a mapped conversation that **CONFIRM** performs in a basic conversation.

APPC Verb Summary

This section briefly describes each APPC verb, grouped by function.

Verbs for Starting Conversations

ALLOCATE or MC_ALLOCATE
: Issued by the local TP. This verb allocates a session between the local logical unit (LU) and a partner LU, and establishes a conversation between the local TP and the partner TP.

  ALLOCATE can establish either a basic or a mapped conversation. **MC_ALLOCATE** can start only a mapped conversation. After the conversation is allocated, APPC uses this verb to return a conversation identifier (**conv_id**).

RECEIVE_ALLOCATE
: Issued by the partner TP. This verb confirms that the partner TP is ready to begin a conversation with the local TP that issued **ALLOCATE** or **MC_ALLOCATE**. Upon successful execution, this verb returns a TP identifier (**tp_id**) for the partner TP and the **conv_id**.

TP_STARTED
: Issued by the local TP. This verb notifies APPC that the local TP is starting. Upon successful execution, this verb returns a **tp_id** for the local TP.

Verbs for Sending Data

CONFIRM or MC_CONFIRM
: Sends the contents of the local LU's send buffer and a confirmation request to the partner TP.

FLUSH or MC_FLUSH
: Flushes the local LU's send buffer, sending the contents of the buffer to the partner LU and TP. If the send buffer is empty, no action takes place.

PREPARE_TO_RECEIVE or MC_PREPARE_TO_RECEIVE
: Changes the state of the conversation from SEND to RECEIVE. Before changing the conversation state, this verb performs the equivalent of **FLUSH**, **MC_FLUSH**, **CONFIRM**, or **MC_CONFIRM**. After this verb has successfully executed, the local TP can receive data.

REQUEST_TO_SEND or MC_REQUEST_TO_SEND
: Informs the partner TP that the local TP wants to send data. The local TP must wait until the partner TP issues

PREPARE_TO_RECEIVE, **MC_PREPARE_TO_RECEIVE**, **RECEIVE_AND_WAIT**, or **MC_RECEIVE_AND_WAIT**, and the conversation state changes to RECEIVE for the partner TP, before the local TP begins sending data.

SEND_DATA or MC_SEND_DATA

Puts data in the local LU's send buffer for transmission to the partner TP.

The data collected in the local LU's send buffer is transmitted to the partner LU and partner TP when one of the following occurs:

- The send buffer fills up.
- The local TP issues FLUSH, MC_FLUSH, CONFIRM, MC_CONFIRM, DEALLOCATE, MC_DEALLOCATE, or another verb that flushes the local LU's send buffer.

Verbs for Receiving Data

POST_ON_RECEIPT or MC_POST_ON_RECEIPT (Microsoft® Windows 2000, Microsoft Windows NT®, Microsoft Windows® 98, and Microsoft Windows® 95 operating systems only)

Issuing this verb allows the application to register to receive a notification when data or status arrives at the local LU without actually receiving it at the same time. This verb can only be issued while in RECEIVE state and it never causes a change in conversation state.

When the TP issues this verb, APPC returns control to the TP immediately. When the specified conditions are satisfied, the Win32® event specified as a parameter is signalled and the verb completes. Then the TP looks at the return code in the verb control block to determine whether or not any data or status notification has arrived at the local LU and issues a **RECEIVE_IMMEDIATE** or **RECEIVE_AND_WAIT** verb to actually receive the data or status notification.

This verb is only supported by Host Integration Server 2000 or by SNA Server 3.0 with Service Pack 1 or later when the APPC applications are running on Windows 2000, Windows NT 4.0, Windows 98, or Windows 95.

RECEIVE_AND_POST or MC_RECEIVE_AND_POST (Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2 operating systems only)

Issuing this verb while the conversation is in RECEIVE state changes the conversation state to PENDING_POST and causes the local TP to receive data asynchronously. This allows the local TP to proceed with processing while data is still arriving at the local LU.

Issuing this verb while the conversation is in SEND state flushes the LU's send buffer and changes the conversation state to PENDING_POST. The local TP then begins to receive data asynchronously.

RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT

Issuing this verb while the conversation is in RECEIVE state causes the local TP to receive any data that is currently available from the partner TP. If no data is available, the local TP waits for data to arrive.

Issuing this verb while the conversation is in SEND state flushes the LU's send buffer and changes the conversation state to RECEIVE. The local TP then begins to receive data.

RECEIVE_IMMEDIATE or MC_RECEIVE_IMMEDIATE

Receives any data that is currently available from the partner TP. If no data is available, the local TP does not wait.

TEST_RTS or MC_TEST_RTS

Determines whether a **REQUEST_TO_SEND** or **MC_REQUEST_TO_SEND** or notification has been received.

Verbs for Confirming Data or Reporting Errors

CONFIRMED or MC_CONFIRMED

Replies to a confirmation request from the partner TP. It informs the partner TP that the local TP has received and processed the data without error.

RECEIVE_LOG_DATA or MC_RECEIVE_LOG_DATA

Issuing this verb allows the user to register to receive the log data associated with an inbound Function Management Header 7 (FMH7) error report. The verb passes a buffer to APPC, and any log data received is placed in that buffer. APPC continues to use this buffer as successive FMH7s arrive until it is provided with another buffer (that is, until the TP issues another **RECEIVE_LOG_DATA** or **MC_RECEIVE_LOG_DATA** specifying a different buffer or no buffer at all).

This verb is only supported by Host Integration Server 2000 or by SNA Server 3.0 with Service pack 1 or later when the APPC applications are running on Windows 2000, Windows NT 4.0, Windows 98, and Windows 95.

SEND_CONVERSATION or MC_SEND_CONVERSATION

Issued by the invoking TP, this verb allocates a session between the local LU and partner LU, sends data on the session, and then deallocates the session.

SEND_ERROR or MC_SEND_ERROR
Notifies the partner TP that the local TP has encountered an application-level error.

Verbs for Getting and Setting Information

GET_ATTRIBUTES or MC_GET_ATTRIBUTES
Used by a TP to get the attributes of the conversation.

GET_LU_STATUS
Used to report the status of a particular remote LU.

GET_STATE
Used by a TP to interrogate the state of a particular conversation.

GET_TP_PROPERTIES
Returns attributes of the TP and the current transaction.

GET_TYPE
Used by a TP to determine the conversation type (basic or mapped) of a particular conversation. With this information, the TP can decide whether to issue basic or mapped conversation verbs.

SET_TP_PROPERTIES
Used to set the attributes of the TP and the current transaction.

Verbs that Provide Management Functions

ACTIVATE_SESSION
Activates a session between the local LU and a specified partner LU, using a specified mode.

This APPC verb is only supported by applications running on Windows 2000, Windows NT, Windows 98, and Windows 95.

CNOS (Change Number of Sessions)
Establishes APPC LU 6.2 session limits.

DEACTIVATE_SESSION
Deactivates a particular session, or all sessions on a particular mode.

This APPC verb is only supported by APPC applications running on Windows 2000, Windows NT, Windows 98, and Windows 95.

DISPLAY
Returns configuration information and current operating values for the SNA node.

Verbs for Ending Conversations

DEALLOCATE or MC_DEALLOCATE
Deallocates a conversation between two TPs. Before deallocating the conversation, this verb performs the equivalent of **FLUSH**, **MC_FLUSH**, **CONFIRM**, or **MC_CONFIRM**.

TP_ENDED
Issued by both the local and partner TPs. It notifies APPC that the TP is ending. Issuing this verb also terminates any active conversations.

Windows APPC Overview

A Windows SNA standard was created to provide one common API to port applications from various operating environments to Microsoft® Windows NT®, Windows® 95, and Windows version 3.*x*. As a direct result of this work, Windows APPC was developed.

The information provided in this guide is source code and executable code compatible with the following implementations of APPC:

- APPC applications based on Host Integration Server 2000 or SNA Server residing on the server or on a client. These applications can be running on Windows 2000, Windows NT, Windows 98, Windows 95, Windows 3.x, MS-DOS®, or OS/2.

- IBM APPC Extended Services for OS/2 version 1.0.

Programs written to use this implementation of APPC can exchange data with programs written to use other implementations of APPC that adhere to the SNA LU 6.2 architecture.

The use of the Windows APPC interface on Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2 will cause additional threads to be created within the calling process. These other threads perform interprocess communication with the Host Integration Server 2000 or SNA Server service over the LAN interface that the client is configured to use (TCP/IP, IPX/SPX, or named pipes, for example).

If an application using Windows APPC is running on Windows 2000 or Windows NT, stopping the SNABASE service will cause the application to be unloaded from memory.

Windows APPC Asynchronous Support

A program that issues a call and does not regain control until the call completes cannot perform any other operations. This type of operation, referred to as blocking, is not suited to a server application designed to handle multiple requests from many clients. Asynchronous call completion returns the initial call immediately so the application can continue with other processes.

Microsoft® Host Integration Server 2000 or SNA Server uses the **RegisterWindowsMessage** function for asynchronous support for APPC applications. With "WinAsyncAPPC" as the input string, an application passes a window handle by which it can be notified of verb completion. The application then issues the verb. When the verb completes, a message is posted to the window handle that was passed, notifying the application that the verb is complete.

With the exception of asynchronous RECEIVE_AND_WAIT, MC_RECEIVE_AND_WAIT, RECEIVE_AND_POST, and MC_RECEIVE_AND_POST, which can issue certain other verbs while pending, a conversation can have only one incomplete operation at any time.

APPC Verbs and Windows Extensions

This topic describes the APPC verbs and Windows extensions that are supported by Host Integration Server 2000 and SNA Server:

APPC Verbs

The following APPC verb descriptions contain important features and should be read before using this version of Windows APPC.

ALLOCATE or MC_ALLOCATE

Issued by the invoking TP, this verb allocates a session between the local LU and partner LU and (in conjunction with RECEIVE_ALLOCATE) establishes a conversation between the invoking TP and the invokable TP. After this verb executes successfully, APPC generates a conversation identifier (**conv_id**). The **conv_id** is a required parameter for all other APPC conversation verbs.

For a user or group using TPs, 5250 emulators, or APPC applications, you can assign default local and remote LUs. In this case, the field for LU alias is left blank or null and the default LUs are accessed when the user or group member starts an APPC program. For more information on using default LUs, see the *Network Integration Services* section of the Microsoft Host Integration Server 2000 online guide.

For Windows version 3.*x* systems, it is recommended that you use WinAsyncAPPC rather than the blocking version of these calls.

RECEIVE_ALLOCATE

Issued by the invokable TP to confirm that it is ready to begin a conversation with the invoking TP that issued ALLOCATE or MC_ALLOCATE. This must be the first APPC verb issued by the invokable TP. The initial state is RESET. If the verb executes successfully (**primary_rc** is AP_OK), the state changes to RECEIVE.

RECEIVE_AND_POST or MC_RECEIVE_AND_POST

Receives application data and status information asynchronously. This allows the local TP to proceed with processing while data is still arriving at the local LU. **RECEIVE_AND_POST** and **MC_RECEIVE_AND_POST** are only supported by the Windows NT, Windows 95, and OS/2 operating systems. For similar functionality under the Windows version 3.*x* system, use **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** in conjunction with WinAsyncAPPC.

While an asynchronous **RECEIVE_AND_POST** or **MC_RECEIVE_AND_POST** is outstanding, the following verbs can be issued:

REQUEST_TO_SEND or **MC_REQUEST_TO_SEND**

GET_TYPE

GET_ATTRIBUTES or **MC_GET_ATTRIBUTES**

TEST_RTS or **MC_TEST_RTS**

DEALLOCATE

SEND_ERROR or **MC_SEND_ERROR**

TP_ENDED

RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT

Receives any data that is currently available from the partner TP. If no data is currently available, the local TP waits for data to arrive. For Windows version 3.*x* systems, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

RECEIVE_AND_WAIT and **MC_RECEIVE_AND_WAIT** have been altered to act like **RECEIVE_AND_POST** and **MC_RECEIVE_AND_POST**. While an asynchronous **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** is outstanding, the following verbs can be issued:

REQUEST_TO_SEND or **MC_REQUEST_TO_SEND**

GET_TYPE

GET_ATTRIBUTES or **MC_GET_ATTRIBUTES**

TEST_RTS or **MC_TEST_RTS**

DEALLOCATE

SEND_ERROR or **MC_SEND_ERROR**

TP_ENDED

TP_STARTED

Issued by the invoking TP, this verb notifies APPC that the TP is starting. For a user or group using TPs, 5250 emulators, or APPC applications, you can assign default local and remote APPC LUs. These default LUs are accessed when the user or group member starts an APPC program (a TP, 5250 emulator, or APPC application) and the program does not specify LU aliases. For more information on using default LUs, see the *Network Integration Services* section of the Microsoft Host Integration Server 2000 online guide.

Windows Extensions

Windows APPC provides a complete set of Windows extensions that allow asynchronous communication using Windows version 3.*x*. These extensions provide maximum programming compatibility between Windows 2000, Windows NT, Windows 98, Windows 95, and Windows version 3.*x*. They include the following:

WinAPPCCancelAsyncRequest

Cancels an outstanding **WinAsyncAPPC**-based request.

WinAPPCCancelBlockingCall

Cancels any outstanding blocking operation for its thread.

WinAPPCCleanup

Terminates and deregisters an application from a Windows APPC implementation. When an application is finished, it must call this function to deregister itself from a Windows APPC implementation.

WinAPPCIsBlocking

Determines if a thread is executing while waiting for a previous blocking call to finish.

WinAPPCSetBlockingHook

Allows a Windows APPC implementation to block APPC function calls by means of a new function. This call is used by Windows version 3.*x* applications to make blocking calls without blocking the rest of the system. Blocking procedures apply only if you do not use asynchronous calls. If a function needs to block, the blocking procedure is called repeatedly until the original request completes. This allows Windows to continue to run while the original application waits for the call to return. Note that while inside the blocking procedure, the application can be re-entered.

WinAPPCSetBlockingHook is used by Windows version 3.*x* applications that go into a **PeekMessageLoop** to make blocking calls without blocking the rest of the system.

By default, Windows NT, Windows 95, and Windows 98 do not go into a **PeekMessageLoop**; they actually block on an event waiting for the call to complete. The only time you need to use **WinAPPCSetBlockingHook** on Windows 2000, Windows NT, Windows 98, or Windows 95 is when a single-threaded application for these systems and Windows version 3.*x* share common source code. In this case, you must explicitly make this call. This is in contrast with the **WinAPPCIsBlocking** and **WinAPPCUnhookBlockingHook** functions.

WinAPPCStartup

Registers an application and specifies the version of Windows APPC required. An application must call this extension to register itself with a Windows APPC implementation before issuing any further Windows APPC calls.

WinAPPCUnhookBlockingHook

Removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

WinAsyncAPPC

Provides an asynchronous version for all of the APPC verbs. It is recommended that you use this extension instead of the blocking versions of the verbs if you run your application under Windows version 3.*x*. APPC verbs that can block are:

ALLOCATE or **MC_ALLOCATE**

CONFIRM or **MC_CONFIRM**

CONFIRMED or **MC_CONFIRMED**

DEALLOCATE or **MC_DEALLOCATE**

FLUSH or **MC_FLUSH**

PREPARE_TO_RECEIVE or **MC_PREPARE_TO_RECEIVE**

RECEIVE_ALLOCATE

RECEIVE_AND_WAIT or **MC_RECEIVE_AND_WAIT**

REQUEST_TO_SEND or **MC_REQUEST_TO_SEND**

SEND_CONVERSATION or **MC_SEND_CONVERSATION**

SEND_DATA or **MC_SEND_DATA**

SEND_ERROR or **MC_SEND_ERROR**

TP_ENDED

TP_STARTED

Limits

Host Integration Server 2000 and SNA Server permit one outstanding Windows APPC asynchronous call per connection and one blocking verb per thread. For example:

```
void ProcessVerbCompletion (WPARAM wParam, LPARAM lParam)
{
    int i;

    for (i = 0; i < nPendingVerbs; i++)
        if (pPendingVerbs[i].hAsync == wParam)
            ProcessVCB( (LPVCB) lParam);
}    . . .

LRESULT CALLBACK SampleWndProc ( ... )
{
    if (msg == uAsyncAPPC) {
        ProcessVerbCompletion(wParam; lParam);
    }
        else switch (msg) {
        case WM_USER:
            if (hAsync = WinAsyncAPPC(hwnd, &vcb))
               pPendingVerbs [nPendingVerbs++].hAsync = hAsync;

            break;
    }
```

```
}

WinMain ( ... )
{
    if ( ( WinAPPCStartup ( ...) = = FALSE ) {
    return FALSE ;

    }

    uAsyncAPPC = RegisterWindowsMessage ("WinAsyncAPPC") ;
    while (GetMessage ( ...)  )  {

    ...
    WinAPPCCleanup ( ... )

}
```

> **Note** The exceptions to the rule of one outstanding asynchronous call are RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT. While these verbs are outstanding, certain other verbs can also be called.

Using APPC Verbs in C Programs

This implementation of APPC is available for programs written in Microsoft® C version 5.1 or later. A C program calls APPC through the external function **APPC**. For compatibility with previous versions of Microsoft C, the external function **APPC_C** is also supported.

> **Note** Compilers other than the Microsoft C compiler can also be used to build applications using this implementation of APPC.

Verb Control Block

The only parameter passed to the **APPC** function is the address of a verb control block (VCB). The VCB is a structure made up of variables that:

- Identify the APPC verb to be executed.
- Supply information to be used by the verb.
- Contain information returned by the verb when execution is complete.

Each APPC verb has its own VCB structure, which is declared in the WINAPPC.H header file. For compatibility with earlier versions, the APPC_C.H header file is also supported.

The WINAPPC.H file is supplied as part of the Host Integration Server 2000 or SNA Server Software Development Kit.

APPC Definition

The prototype definitions of the **APPC** function are as follows:

```
void WINAPI APPC(long);
HANDLE WINAPI WinAsyncAPPC (hWnd, LPAPPC);
```

The VCB address parameter, a 32-bit pointer, is declared as a long integer and thus requires casting from a pointer to a long integer.

Issuing an APPC Verb

The following procedure is required to issue a blocking APPC verb. In the sample code, the verb issued is **MC_SEND_DATA**.

To issue a blocking APPC verb

1. Create a structure variable from the VCB structure that applies to the APPC verb to be issued.

```
#include <winappc.h>
    .
    .
struct mc_send_data  mcsend;

The VCB structures are declared in WINAPPC.H; one of these structures is:
mc_send_data
```

2. Clear (set to zero) the variables within the VCB structure.

```
memset( mcsend, '\0', sizeof( mcsend ) );
```

3. Assign values to the VCB variables that supply information to APPC.

```
mcsend.opcode = AP_M_SEND_DATA;
mcsend.opext = AP_MAPPED_CONVERSATION;
memcpy( mcsend.tp_id, tp_id, sizeof( tp_id ) );
mcsend.conv_id = conv_id;
mcsend.dlen = datalen;
mcsend.dptr = sharebufptr;
```

4. The values AP_MAPPED_CONVERSATION and AP_M_SEND_DATA are symbolic constants representing integers. These constants are defined in WINAPPC.H.

5. Invoke the **APPC** function. The only parameter is a pointer to the address of the structure containing the VCB for the desired verb.

```
APPC ( ( long ) (void FAR * ) &mcsend );
```

6. Use WinAsyncAPPC if you are running the application under Windows version 3.*x*.

7. To call **WinAsyncAPPC**:

8.
```
HANDLE WINAPI WinAsyncAPPC (hWnd, lpVCB)
```

9. When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinAsyncAPPC" as the input string.
10. Use the variables that were returned by APPC.

```
if( mcsend.primary_rc != AP_OK )
/* Do error routine */
    .
    .
    .
```

Windows 2000, Windows NT, Windows 98, and Windows 95 Considerations

This topic summarizes information about developing transaction programs (TPs) using APPC for the Microsoft® Windows® 2000, Windows NT®, Windows 98, and Windows 95 operating systems.

Host Integration Server 2000 with Service Pack 1 runs on the following additional operating systems:

- Microsoft Windows XP Professional

- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

Byte ordering

The values of constants defined in WINAPPC.H and WINCSV.H are dependent on the byte ordering of the hardware used. Macros are used to set the constants to the correct value.

By default, Intel little-endian byte ordering is used, with the low byte of a 16-bit value followed by the high byte. However, when defining inline macros, the NON_INTEL_BYTE_ORDER macro used in WINAPPC.H and WINCSV.H will not reverse (flip) the byte order for constants. Non-constant input parameters in VCBs (such as lengths, pointers, and so on) are always in the native format.

For example, the primary return code of AP_PARAMETER_CHECK is defined to have a value of 0x0001. Depending on the environment (byte ordering), the constant AP_PARAMETER_CHECK may or may not be 0x0001. Some formats define the value as it appears in memory; others define it as a 2-byte variable. Because you cannot assume that the application will always use provided constants rather than hardwired values, you can define a macro to swap the bytes. The following is an example of using the macro:

```
/* when NON_INTEL_BYTE_ORDER is specified, the APPC_FLIPI macro defined in WINAPPC.H macro b
ecomes */
#define APPC_FLIPI(x)    (x)

/* otherwise this macro flips bytes by defining */
#define APPC_FLIPI(X) APPC_MAKUS(APPC_HI_UC(X),APPC_LO_UC(X))

/* the AP_PARAMETER_CHECK macro is now defined using the APPC_FLIPI macro */
#define AP_PARAMETER_CHECK APPC_FLIPI (0X0001)      /* X '0001' */
```

Events

To receive data asynchronously, an event handle is passed in the semaphore field of the VCB. This event must be in the non-signaled state when passed to APPC, and the handle must have EVENT_MODIFY_STATE access to the event.

Library names

In order to support the coexistence of Win16 and Win32® API libraries on the same computer, the Win32 DLL names have been changed.

Old DLL names	New DLL names
WINAPPC.DLL	WAPPC32.DLL
WINCSV.DLL	WINCSV32.DLL

The old DLL names should be used for Win32-based applications that are required to run on SNA Server version 2.0. The new DLL names should be used for Win32-based applications that are intended to run only on SNA Server version 2.1 or later and Host Integration Server 2000.

If you intend your Win32-based application to be used with SNA Server version 2.0, you should link with the libraries included with SNA Server version 2.0. Otherwise, use the new libraries provided with SNA Server versions 2.1, 3.0, and 4.0 and provided with Host Integration Server 2000.

Limits

For Windows 2000, Windows NT, Windows 98, and Windows 95, the number of simultaneous CSVs allowed per process is 64. Only one of these verbs per thread can be synchronous (blocking).

Using APPC, the maximum number of simultaneous conversations per process is 15,000. Each process supports up to 15,000 simultaneous TPs.

Multiple threads

A TP can have multiple threads that issue verbs. Windows APPC makes provisions for multithreaded Windows-based processes. A process contains one or more threads of execution. All references to threads refer to actual threads in the multithreaded Windows 2000, Windows NT, Windows 98, and Windows 95 environments.

With the exception of RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT, only one conversation verb can be outstanding at a time on any conversation; however, other verbs can be issued for other conversations. This guideline also applies to TP verbs and TPs. Although multiple TP verbs can be issued, only one TP verb can be outstanding at a time on a TP. This applies to both multithreaded applications and single-threaded applications that use asynchronous calls.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or DWORD (4-byte) boundaries, whichever is smaller. As a result, DWORDs are aligned on DWORD boundaries, WORDs are aligned on WORD boundaries, and BYTEs are aligned on BYTE boundaries. This means, for example, that there is a 2-byte gap between the primary and secondary return codes. Therefore, the elements in a VCB should only be accessed using the structures provided.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Registering and deregistering applications

All Windows APPC applications must call WinAPPCStartup at the beginning of the session to register the application and WinAPPCCleanup at the end of the session to deregister the application.

All Windows CSV applications must call the Windows SNA extension WinCSVStartup at the beginning of the session to register the application and WinCSVCleanup to deregister the application when the session is finished.

Run-time linking

For a TP to be dynamically linked to APPC at run time, the TP must issue the following calls:

- LoadLibrary to load the dynamic-link libraries WINAPPC.DLL or WAPPC32.DLL.
- GetProcAddress to specify APPC on all the desired entry points to the DLL such as APPC, WinAsyncAPPC, WinAPPCStartup, and WinAPPCCleanup.

For a TP to be dynamically linked to CSV at run time, the TP must issue the following calls:

- LoadLibrary to load WINCSV.DLL or WINCSV32.DLL, the dynamic-link libraries for Windows CSV.
- GetProcAddress to specify CSV on all the desired entry points to the DLL such as ACSSVC, WinAsyncCSV, WinCSVStartup, and WinCSVCleanup.

The TP must issue the **FreeLibrary** call when the APPC or CSV library is no longer required.

Yielding to other components

Because the Windows 2000, Windows NT, Windows 98, and Windows 95 environments are multithreaded, there is no need to yield to other components. However, if an application is single-threaded and is intended for both Windows version 3.*x* and these other operating systems, use the Windows extensions WinAPPCSetBlockingHook and WinAPPCUnhookBlockingHook. These extensions must be explicitly called to accomplish the yield procedure.

Windows 3.x Considerations

This topic summarizes information about developing TPs for the Microsoft® Windows® version 3.*x* system.

Blocking routines

Do not use blocking functions if your application runs in Windows version 3.*x*. Instead, use WinAsyncAPPC in conjunction with a **WinAsyncAPPC** Windows message.

Limits

For the Windows version 3.x system, the number of simultaneous CSVs allowed per process is 64. Only one of these verbs per process can be synchronous.

Using APPC, the maximum number of simultaneous conversations per process is 64. Each process supports up to 16 simultaneous TPs.

Load-time linking

For a TP to be dynamically linked to APPC at load time, you must do one of the following at link time:

- Insert the following IMPORTS statements in the definition (.DEF) file used to link the TP:

```
IMPORTS WINAPPC.APPC
IMPORTS WINAPPC WinAsyncAPPC.APPC
```

- Link the TP to WINAPPC.LIB, which contains the entry point linkage information for APPC. If you intend to use CSVs, you must also link to WINCSV.LIB, which contains the entry point information for CSVs.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDs are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

RECEIVE_AND_POST and MC_RECEIVE_AND_POST

These verbs have been replaced for Windows version 3.*x* by calling RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT using WinAsyncAPPC.

Registering and deregistering applications

All Windows APPC applications must call WinAPPCStartup at the beginning of the session to register the application and WinAPPCCleanup at the end of the session to deregister the application.

All Windows CSV applications must call the Windows SNA extension WinCSVStartup at the beginning of the session to register the application and WinCSVCleanup to deregister the application when the session is finished.

Run-time linking

For a TP to be dynamically linked to APPC at run time, the TP must issue the following calls:

- **LoadLibrary** to load WINAPPC.DLL, the dynamic-link library for APPC.
- **GetProcAddress** to specify APPC on all the desired entry points to the DLL such as APPC, WinAsyncAPPC, WinAPPCStartup, and WinAPPCCleanup.

The TP must issue the **FreeLibrary** call when the APPC library is no longer required.

Simultaneous conversations

A TP can participate in as many as 64 conversations simultaneously within the Windows 3.x environment. However, if more than one TP is active at once, the total number of conversations cannot exceed 64.

Translating service TP names to ASCII for WIN.INI

For service TPs on Host Integration Server 2000 or SNA Server clients running Windows version 3.*x*, a line must be added to the WIN.INI file, specifying the TP name in ASCII. For more information, see Translating SNA Service TP Names to ASCII for WIN.INI.

Yielding to other components

Because Windows version 3.*x* is single-threaded, there is only one thread of execution. In the case where a function must wait before completing a task, the only thread of execution could block to allow other tasks to proceed.

This means that while in a blocking call, the calling application's Window procedure can be called. To test if this is the case, the extension WinAPPCIsBlocking is provided. Any attempt to make a second blocking call with one already outstanding will cause the call to fail with the return code AP_THREAD_BLOCKING.

Windows APPC and CSV contain a default yield procedure for Windows version 3.*x* that can yield to other functions while waiting for the first function to complete. The default is:

```
BOOL DefaultBlockingHook (void)  {
    MSG msg:
    /* get the next message if any */
    if ( PeekMessage (&msg,0,0,PM_NOREMOVE)  )   {
if  ( msg.message = = WM_QUIT  )
    return FALSE;   / /  let app process WM_QUIT
PeekMessage  ( &msg,0,0,PM_REMOVE)  ;
TranslateMessage  (&msg)  ;
DispatchMessage  (&msg)  ;
    }
    /*TRUE if no WM_QUIT received */
    return TRUE;
}
```

Besides the default yield procedure, Windows APPC provides WinAPPCSetBlockingHook to support applications that require more complex message processing. This call allows a Windows APPC implementation to block APPC function calls by means of a new function. It is used by Windows version 3.*x* applications to make blocking calls without blocking the rest of the system. To call **WinAPPCSetBlockingHook:**

```
FARPROC WINAPI WinAPPCSetBlockingHook (FARPROC lpBlockFunc)
```

WinAPPCUnhookBlockingHook removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism. To call **WinAPPCUnhookBlockingHook**:

```
BOOL WINAPI WinAPPCUnhookBlockingHook (void)
```

MS-DOS Considerations

This topic summarizes information about developing TPs for the Microsoft® MS-DOS® system.

AS/400 PC Support Router API

MS-DOS support for Windows APPC is handled in one of two ways:

- The call-level interface through the DOSACS.LIB library provided with the Host Integration Server 2000 or SNA Server Software Development Kit
- The software interrupt PC Support Router API

For additional information on the PC Support Router API, see the *IBM PC Support/400 Application Program Interface Reference*, part number SC41-8254.

CSV restrictions when SnaBase is not loaded

If your application uses the Windows-based client or an MS-DOS-based client supplied with Host Integration Server 2000 or SNA Server, and SnaBase has not been loaded, only the following verbs are available:

GET_CP_CONVERT_TABLE

CONVERT

Under these circumstances, **CONVERT** cannot use a type G conversion table. Attempting to use a type G conversion table causes the system to return the SV_TABLE_ERROR primary return code.

Attempting to issue COPY_TRACE_TO_FILE, DEFINE_TRACE, LOG_MESSAGE, or TRANSFER_MS_DATA causes the system to return the SV_COMM_SUBSYSTEM_NOT_LOADED primary return code.

Limits

For the MS-DOS system, only one outstanding CSV verb is allowed per process. Only one outstanding verb per process can be synchronous.

Using APPC, the maximum number of simultaneous conversations per process is 16. Each process supports up to four simultaneous TPs.

Link library

The link library for APPC is DOSACS.LIB.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDs are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Partner TPs

Host Integration Server 2000 or SNA Server invokable TPs cannot be run on MS-DOS-based clients. No support is provided for autostarting MS-DOS partner TPs. The call-level interface contains the same syntax as that of OS/2.

RECEIVE_AND_POST and MC_RECEIVE_AND_POST

These verbs are not available with MS-DOS.

Simultaneous conversation
In an MS-DOS environment, a TP can participate in as many as 16 conversations simultaneously.

SnaBase
When running a TP on a Host Integration Server 2000 or SNA Server MS-DOS-based client, you must use SnaBase, which supports APPC. If you attempt to use something other than SnaBase, your TP will not run.

OS/2 Considerations

This topic summarizes information about developing TPs for the OS/2 system.

Critical sections
Exercise great caution when using critical sections, which are the parts of a program that must run without interruption. A TP must not issue an APPC verb within a critical section.

Data segments
Data is sent from and received in data buffers established by the TP. A data buffer must reside on an unnamed shared data segment and it must fit entirely within the data segment. Many data buffers can reside on the same data segment.

To allocate a data segment, use the **DosAllocSeg** function with *Flags* equal to 1.

To improve efficiency, a TP can reuse the same data segment each time it issues a verb requiring a data buffer. If necessary, the program can allocate a segment of up to 64K and then partition the segment into data buffers.

Limits
For the OS/2 system, only one outstanding CSV verb is allowed per process. Only one outstanding verb per process can be synchronous.

Using APPC, the maximum number of simultaneous conversations per process is 64. Each process supports up to 64 simultaneous TPs.

Load-time linking
For a TP to be dynamically linked to APPC at load time, you must do one of the following at link time:

- Insert the following IMPORTS statement in the definition (.DEF) file used to link the TP:

```
IMPORTS APPC.APPC
```

- Link the TP to APPC.LIB, which contains the entry point linkage information for various APIs, including APPC.

Multiple processes
Multiple processes cannot have the same TP identifier (**tp_id**). Only the process that issues TP_STARTED or RECEIVE_ALLOCATE can use the **tp_id** returned by the verb. Another process that needs to use APPC must issue **TP_STARTED** or **RECEIVE_ALLOCATE** to obtain its own **tp_id**.

Two or more instances of the same TP can be run as different processes, but each will be assigned a different **tp_id**.

One process can contain many TPs, each with its own **tp_id**. In this case, you may want to create a separate thread for each TP to avoid the possibility of a deadlock. (A deadlock occurs when an APPC verb that is hung blocks the execution of verbs in other conversations and TPs.)

A process containing many TPs can issue two or more verbs simultaneously, provided that each verb is for a different TP (specifies a different **tp_id**).

Multiple threads
A TP can have multiple threads that issue verbs. Windows APPC makes provisions for multithreaded processes. A process contains one or more threads of execution. All references to threads refer to actual threads in the multithreaded OS/2 environment.

With the exception of RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT, only one conversation verb can be outstanding at a time on any conversation; however, other verbs can be issued for other conversations. This guideline also applies to TP verbs and TPs. Although multiple TP verbs can be issued, only one TP verb can be outstanding at a time on a TP. This applies to both multithreaded applications and single-threaded applications that use asynchronous calls.

Packing
For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned

on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDs are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Run-time linking

For a TP to be dynamically linked to APPC at run time, the TP must issue the following calls:

- DosLoadModule to load APPC.DLL, the DLL for APPC.
- DosGetProcAddr to specify APPC as the desired entry point to the DLL.

Unlinking (the **DosFreeModule** call) is not supported.

Simultaneous conversations

A TP can simultaneously participate in as many as 64 conversations for each OS/2 process.

SNA Server or Host Integration Server CSVs

Applications running on Host Integration Server 2000 or SNA Server using common service verbs (CSVs) are compatible with the common services programming interface provided by IBM ES for OS/2 version 1.0, with the following exceptions:

- VCBs are not packed. As a result, DWORDs and WORDs are on WORD boundaries, and BYTEs are on BYTE boundaries. This means, for example, that there is not a 2-byte gap between the primary and secondary return codes. VCBs should be accessed using the structures provided, and compiler options that change this packing method should be avoided.
- Trace information for products is stored in trace files, not in a storage buffer. With the Host Integration Server 2000 or SNA Server implementation of COPY_TRACE_TO_FILE_sna_COPY_TRACE_TO_FILE_appc, the trace information in these files is copied to a single trace file.
- The additional tracing features of DEFINE_TRACE_sna_DEFINE_TRACE_appc are not applicable because of differences between the architecture of the Host Integration Server 2000 and SNA Server products and the architecture of IBM Communications Manager. The tracing features provided in IBM OS/2 ES version 1.0 include event tracing, automatic tracing, and tracing support for the following:

X.25 API verbs, frame data, and data link control (DLC) data
Twinaxial data

- With the Host Integration Server 2000 or SNA Server implementation of GET_CP_CONVERT_TABLE, user-defined code pages are applicable only in an OS/2 environment. These code pages are supported for only the Windows NT, Windows 95, and OS/2 systems because of the additional memory occupancy in the MS-DOS and Windows environments. The new user-defined code pages are explained in the description of the verb.
- With the Host Integration Server 2000 and SNA Server implementation of TRANSFER_MS_DATA, support for the PD_STATS subvector type is retained.
- **DEFINE_DUMP** and **SET_USER_LOG_QUEUE**, provided in IBM OS/2 ES version 1.0, are not applicable because of differences between the architecture of Host Integration Server 2000 and SNA Server and the architecture of IBM Communications Manager. If either verb is called, INVALID_VERB is returned.

Stack size

The recommended stack size for a TP is at least 3000 bytes.

When executing a verb, APPC uses the calling TP's stack. The combination of OS/2 and APPC requires 2560 bytes of stack space, and the TP requires additional stack space for its variables.

Translating service TP names to ASCII for SNA.INI

For service TPs on Host Integration Server 2000 or SNA Server clients running OS/2, a line must be added to the SNA.INI file, specifying the TP name in ASCII. For more information, see Translating SNA Service TP Names to ASCII for SNA.INI.

VCB segment

The segment containing the VCB must be a writable segment.

About Transaction Programs

A processing task accomplished by programs using APPC is called a transaction. Consequently, programs that use APPC are called transaction programs, or TPs. These programs communicate as peers, on an equal (rather than hierarchical) basis. The TPs use APPC verbs to exchange status information and application data. Each TP uses APPC verbs to supply parameters to APPC, which performs the desired function and returns parameters to the TP.

TPs distributed across a local or wide area network perform distributed transaction processing.

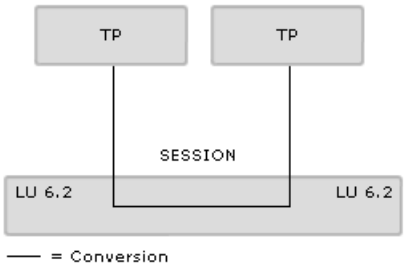
This section describes how to write TPs and how to configure the systems on which TPs run. The topics in this section cover the following general areas:

- Understanding fundamental concepts related to TPs
- Designing and coding TPs
- Configuring registry and environment variables for invokable TPs
- Configuring Microsoft® Host Integration Server 2000 or Microsoft SNA Server 4.0 to work with your TPs
- Sync Point Level 2 support

This section contains:

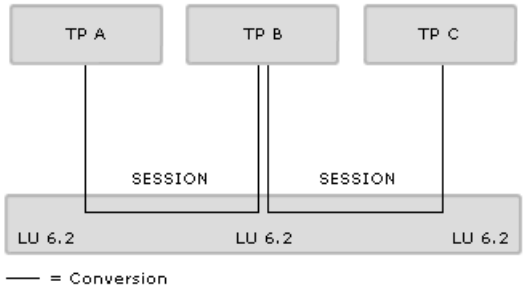
Communication Between TPs

Various hardware and software elements in the SNA environment are required for two TPs to communicate with each other. The following figure shows several fundamental elements.



Each TP is associated with a logical unit (LU) of type 6.2. The LU allows the TP to access the network. Several TPs can be associated with the same LU.

A partner TP can invoke another TP which, in turn, invokes another TP, and so on. In the following figure, TP A invokes TP B and TP B invokes TP C.



This section contains:

- Fundamental Terms for TPs and LUs
- Sample TPs Illustrating Fundamental Concepts
- Configuring and Controlling TPs
- Creating TPs and Their Supporting Configuration

Fundamental Terms for TPs and LUs

The following terms describe some fundamental characteristics of TPs communicating through LUs:

asynchronous verb
An APPC verb for which the initial function call returns immediately, so that the normal operation of the program is not blocked while processing of the verb completes. For more information, see Receiving Data Asynchronously.

basic conversation
A type of conversation more complex than a mapped conversation and generally used by service TPs (SNA-based programs that provide services to other programs). For more information, see Basic and Mapped Conversations Compared.

conversation
The interaction between TPs carrying out a specific task. Each conversation requires an LU-LU session. A TP can be involved in several conversations simultaneously, as shown with TP B in Communication Between TPs.

invokable TP
A TP that can be invoked by another TP. Invokable TPs are usually server-type applications; that is, they work in the same general way that an application such as CICS works. Parameters for an invokable TP are configured through registry or environment variables.

There are several types of invokable TPs:

operator-started invokable TP

A TP that is started manually in preparation for being invoked.

autostarted invokable TP

A TP that is automatically started by APPC when invoked.

queued TP

A TP that, when invoked multiple times, loads once and then queues up subsequent requests to be dealt with one at a time. All operator-started TPs and some autostarted TPs are queued.

nonqueued TP

A TP loaded multiple times, once for every time it is invoked. Some autostarted TPs are nonqueued but no operator-started TPs are nonqueued.

For more information, see Invokable TPs.

invoking TP
A TP that can invoke (that is, initiate a conversation with) other TPs. Invoking TPs are usually client-type applications; that is, they work in the same general way that an emulator works. For more information, see Invoking TPs.

local LU and **local TP**
An LU and TP working together, when viewed as the "home base" for a particular conversation. From this viewpoint, some other LU and TP are seen as the "partner" or "remote" LU and TP.

LU alias
The string that identifies an LU to a TP. The alias can be up to eight characters long.

LU-LU session
The communication between two LUs over a specific connection for a specific amount of time. An LU-LU session is needed for two TPs to interact. One session can be used serially by many pairs of TPs.

An LU 6.2 can have multiple sessions (two or more concurrent sessions with different partner LUs) and parallel sessions (two or more concurrent sessions with the same partner LU).

LUs as well as LU-LU pairs and modes are configured using the SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0.

mapped conversation
A type of conversation simpler than a basic conversation and generally used by application TPs (programs that accomplish tasks for end users). The characters **MC_** at the beginning of a verb stand for mapped conversation. For more information, see Basic and Mapped Conversations Compared.

partner LU and **partner TP**, or **remote LU** and **remote TP**
An LU and TP working together, when viewed as being at the far end of a particular conversation.

synchronous verb
An APPC verb that blocks further program operations until the processing of the verb is complete.

Sample TPs Illustrating Fundamental Concepts

A set of sample TPs is provided on the Host Integration Server and on the SNA Server CD-ROM in the \SDK\SAMPLES directory. Included with the sample code in the \SDK\SAMPLES\SNA\TPSETUP directory on the Host Integration Server CD-ROM is TPSETUP, a program that simplifies the setting of registry or environment variables needed by autostarted invokable TPs. Without an interface like that provided by TPSETUP, configuring such variables can be complicated and error-prone. Therefore, it is recommended that you use code like TPSETUP in installation programs for autostarted invokable TPs.

The source code for TPSETUP (INSTALL.C) can be compiled to work in the Microsoft Windows® 2000, Windows NT®, Windows 98, Windows 95, or Windows version 3.*x* environment.

For information about TPSETUP and about the sample TPs, see Sample APPC TPs in the SDK.

Configuring and Controlling TPs

The following table shows how the characteristics of the TPs and selection of the LUs for a conversation are controlled.

Characteristic	How controlled
Type of verb: synchronous or asynchronous	Written into the code. Synchronous verbs use blocking calls; asynchronous verbs avoid blocking calls. See Receiving Data Asynchronously and WinAsyncAPPC.
Type of conversation: basic or mapped	Written into the code. The **MC_** prefix is used on verbs in mapped conversations and omitted on verbs in basic conversations. For two TPs to communicate successfully, both must use the same type of conversation, basic or mapped. See Basic and Mapped Conversations Compared.
Type of TP: invoking or invokable	Written into the code. Invoking TPs start with TP_STARTED, which identifies the invoking TP, and ALLOCATE or MC_ALLOCATE, which identifies the requested invokable TP. Invokable TPs start with RECEIVE_ALLOCATE, which identifies the invokable TP. See Invoking TPs and Invokable TPs.
The local LU alias to be used by an invoking TP	Three options: • Written into the code in TP_STARTED. • Configured (in Host Integration Server Manager or SNA Explorer) as the default local APPC LU for the user who starts the invoking TP. • Configured as a member of the default outgoing local APPC LU pool using the SNA Manager on Host Integration Server 2000 and using SNA Explorer on SNA Server 4.0 See Configuring Invoking TPs on Host Integration Server and SNA Server.
The invokable TP requested by an invoking TP	Written into the ALLOCATE or MC_ALLOCATE request in the invoking TP.
The LU alias to be used by an invokable TP	Two options: • Written into the invoking TP (not the invokable TP), in ALLOCATE or MC_ALLOCATE. • Configured as the default remote APPC LU for the user who starts the invoking TP. See Configuring Invoking TPs on Host Integration Server and SNA Server and Matching Invoking and Invokable TPs.
Type of autostarted invokable TP: queued or nonqueued	Configured with registry or environment variables. See Configuring Invokable TPs.
Local LU and remote LU aliases	Configured using SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0. For information, see the *Microsoft Host Integration Server 2000 online books*.
The pairing of local and remote LUs, and the mode used for each LU-LU pair	Configured using SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0. For information, see the *Microsoft Host Integration Server 2000 online books*.

Creating TPs and Their Supporting Configuration

The following procedure describes how to create TPs and set up a supporting configuration.

To create TPs and set up a supporting configuration

1. Write, compile, and link each TP.
2. Place each TP on an appropriate computer.

   For TPs that you will start many times or that will be started by a user, arrange for the TP to be started easily. That is, for graphical interfaces, create a program icon for starting the TP; for non-graphical interfaces, make sure the TP is in the path.

3. On one or more servers running Host Integration Server or SNA Server, configure LUs, modes, and LU-LU pairs for use by the TPs.

   For information about how to set up LU-LU pairs to support TPs, see Using Invoking and Invokable TPs and the *Microsoft Host Integration Server 2000 online books*.

4. Set any registry or environment variables needed for the invokable TP.

   For autostarted invokable TPs, it is recommended that you use the sample TP configuration program, TPSETUP, for this step. When you write an installation program for autostarted invokable TPs, it is recommended that you include code similar to TPSETUP.

   For information about registry or environment variables, see Configuring Invokable TPs. For information about TPSETUP, see Sample APPC TPs in the SDK.

5. If the invokable TP is operator-started, start it, or arrange for it to be started when the computer is restarted and then restart the computer.

   If the invokable TP is autostarted, Host Integration Server 2000 will start it when needed.

6. Start the invoking TP.

Designing and Coding TPs

The following topics provide background information about designing and coding TPs.

This section contains:

Conversation States

The state of the conversation (as viewed by a particular TP) governs which APPC verbs can be issued by the TP at a particular time. For example, a TP cannot issue MC_SEND_DATA if the conversation is not in SEND state for that TP.

The state of a conversation depends on the TP from which it is viewed. A local TP can view a conversation as being in SEND state while the partner TP views the conversation as being in RECEIVE state. A particular TP can be in several conversations, each of which is in a different state.

The possible conversation states are summarized here.

CONFIRM
The TP has received a request for confirmation of receipt of data; it must respond positively or send error information to the partner TP.

CONFIRM_DEALLOCATE
The TP has received a request for confirmation; it must respond positively or send error information. If the TP responds positively, the conversation is automatically deallocated.

CONFIRM_SEND
The TP has received a request for confirmation; it must respond positively or send error information. After responding, the TP can begin to send data.

PENDING_POST
The TP is receiving data asynchronously. The TP can perform other processing not related to this conversation.

RECEIVE
The TP can receive application data and status information from the partner TP. When the conversation is in RECEIVE state, the TP can also send error information and request permission to send data.

RESET
The conversation has not started or has been terminated.

SEND
The TP can send data to the partner TP and request confirmation. When the conversation is in SEND state, the TP can also begin to receive data, which changes the state to RECEIVE.

SEND_PENDING
The TP issued a receive verb and the **what_rcvd** parameter returned by that verb indicated both data received and a status indication of SEND. This only affects the use of the **err_dir** parameter for SEND_ERROR and MC_SEND_ERROR_sna_MC_SEND_ERROR_appc. Otherwise, the state is the same as the SEND state.

This section contains:

- State Checks
- Changing Conversation States

State Checks

A state check occurs when a TP issues an APPC verb and the conversation is not in the appropriate state. For example, a state check occurs if a TP issues MC_SEND_DATA_sna_MC_SEND_DATA_appc while the conversation is in RECEIVE state. When a state check occurs, APPC does not execute the verb; it returns state check information through primary and secondary return codes.

Changing Conversation States

A change in the conversation state can result from:

- A verb issued by the local TP.
- A verb issued by the partner TP.
- An error condition.

The following example shows how APPC verbs can change the state of the conversation from SEND to RECEIVE and from RECEIVE to SEND.

> Any TP can send or receive data, regardless of whether it is the invoking TP (the TP that started the conversation) or the invokable TP (the TP that responded to a request to start a conversation).

This example shows how APPC verbs can change the conversation state. In this table, each conversation state appears in bold and precedes the APPC verbs that are used while in that state.

Issued by the invoking TP	**Issued by the invokable TP**
TP_STARTED	
Conversation state: RESET	
MC_ALLOCATE	
(synclevel=AP_CONFIRM_SYNC_LEVL)	
Conversation state: SEND	
MC_SEND_DATA	
MC_PREPARE_TO_RECEIVE	
(ptr_type=AP_SYNC_LEVEL)	
	Conversation state: RESET
	RECEIVE_ALLOCATE
	Conversation state: RECEIVE
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(what_rcvd=AP_DATA_COMPLETE)
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(what_rcvd=AP_CONFIRM_SEND)
	Conversation state: CONFIRM_SEND
	MC_CONFIRMED
	Conversation state: SEND
	MC_SEND_DATA
	MC_CONFIRM
Conversation state: RECEIVE	
MC_RECEIVE_AND_WAIT	
(primary_rc=AP_OK)	
(what_rcvd=AP_DATA_COMPLETE)	
MC_RECEIVE_AND_WAIT	
(primary_rc=AP_OK)	
(what_rcvd=AP_CONFIRM_WHAT_RECEIVED)	
Conversation state: CONFIRM	
MC_REQUEST_TO_SEND	
MC_CONFIRMED	
	(rts_rcvd=AP_YES)
	MC_PREPARE_TO_RECEIVE
	(ptr_type=AP_SYNC_LEVEL)
Conversation state: RECEIVE	
MC_RECEIVE_AND_WAIT	
(primary_rc=AP_OK)	

(what_rcvd=AP_CONFIRM_SEND)	
Conversation state: CONFIRM_SEND	
MC_CONFIRMED	
Conversation state: SEND	
MC_SEND_DATA	
MC_DEALLOCATE	
(dealloc_type=AP_SYNC_LEVEL)	
	Conversation state: RECEIVE
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(what_rcvd=AP_DATA_COMPLETE)
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(what_rcvd=AP_CONFIRM_DEALLOCATE)
	Conversation state: CONFIRM_DEALLOCATE
	MC_CONFIRMED
Conversation state: RESET	**Conversation state: RESET**
TP_ENDED	TP_ENDED

Initial States

Before the conversation is allocated, the state is RESET for both TPs.

In the example, after the conversation is allocated, the initial state is SEND for the invoking TP and RECEIVE for the invokable TP.

Changing to RECEIVE State

MC_PREPARE_TO_RECEIVE allows a TP to change the conversation from SEND to RECEIVE state. This verb:

- Flushes the local LU's send buffer.
- Sends the AP_CONFIRM_SEND indicator to the partner TP through the what_rcvd parameter of a receive verb. This indicator tells the partner TP that an MC_CONFIRMED response is expected before the partner TP can begin to send data.

Confirmation processing is performed when the following conditions are true:

- The **ptr_type** parameter is set to AP_SYNC_LEVEL.
- The synchronization level of the conversation is set to AP_CONFIRM_SYNC_LEVEL.

For more information about confirmation processing, see Confirmation Processing.

> Issuing MC_RECEIVE_AND_WAIT while the conversation is in SEND state flushes the LU's send buffer and changes the conversation state to RECEIVE. Changing the conversation state in this manner does not support confirmation processing.

Changing to SEND State

MC_REQUEST_TO_SEND informs the partner TP (for which the conversation is in SEND state) that the local TP (for which the conversation is in RECEIVE state) wants to send data. This request is communicated to the partner TP through the **rts_rcvd** parameter of MC_CONFIRM. (The **rts_rcvd** parameter is also returned to MC_SEND_DATA and other verbs.)

When the partner TP issues MC_PREPARE_TO_RECEIVE, the conversation state changes to RECEIVE for the partner TP, making it possible for the local TP to send data.

> Issuing MC_REQUEST_TO_SEND does not change the state of the conversation. Upon receiving a request to send, the partner TP is not required to change the conversation state; it can ignore the request.

Confirmation Processing

The sequence of events for confirmation processing is as follows:

1. Establish the synchronization level.
2. Send a confirmation request.
3. Receive data and confirmation request.
4. Respond to the confirmation request.
5. Deallocate the conversation.

Using confirmation processing, a TP sends a confirmation request with the data; the partner TP confirms receipt of the data or indicates that an error occurred. Each time the two TPs exchange a confirmation request and response, they are synchronized.

> Although the example in this section does not show this, any TP can send or receive data, regardless of whether the TP is the invoking TP or the invokable TP.

The following example illustrates confirmation processing.

Issued by the invoking TP	**Issued by the invokable TP**
TP_STARTED	
MC_ALLOCATE	
(synclevel=AP_CONFIRM_SYNC_LEVEL)	
MC_SEND_DATA	
(type=AP_SEND_DATA_CONFIRM)	
	RECEIVE_ALLOCATE
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(rtn_status=AP_YES)
	(what_rcvd= AP_DATA_COMPLETE_CONFIRM)
	MC_CONFIRMED
MC_SEND_DATA	
(type=AP_SEND_DATA_DEALLOC_SYNC_LEVEL)	
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(rtn_status=AP_YES)
	(what_rcvd= AP_DATA_COMPLETE_CONFIRM_ DEALLOCATE)
	MC_CONFIRMED
TP_ENDED	TP_ENDED

Establishing the Synchronization Level

The **synclevel** parameter of MC_ALLOCATE determines the synchronization level of the conversation. There are three possible synchronization levels:

- AP_NONE, under which confirmation processing does not occur.
- AP_CONFIRM_SYNC_LEVEL, under which the TPs can request confirmation of receipt of data and respond to requests for confirmation of data.
- AP_SYNCPT, under which the TPs operate under Sync Point Level 2 support for confirmation of receipt of data.

Sending a Confirmation Request

MC_SEND_DATA with type AP_SEND_DATA_CONFIRM has two effects:

- It flushes the local LU's send buffer and sends any data contained in the buffer to the partner TP.
- It sends a confirmation request that the partner TP receives through the what_rcvd parameter of a receive verb.

After issuing **MC_SEND_DATA**, the local TP waits for confirmation from the partner TP.

Receiving Data and Confirmation Request

The **what_rcvd** parameter of MC_RECEIVE_AND_WAIT indicates:

- Status of the data received: complete or incomplete.
- Future processing expected of the local TP.

In the example, **what_rcvd** is AP_DATA_COMPLETE_CONFIRM, indicating that the status is complete and a confirmation is requested.

Responding to a Confirmation Request

The partner TP issues MC_CONFIRMED to confirm receipt of data. This frees the local TP to resume processing.

Deallocating the Conversation

MC_SEND_DATA sends a confirmation request with the data when all of the following conditions are true:

- The conversation's synchronization level (established by the **synclevel** parameter of MC_ALLOCATE) is AP_CONFIRM_SYNC_LEVEL.
- The type parameter of MC_SEND_DATA is set to AP_SEND_DATA_DEALLOC_SYNC_LEVEL.
- The what_rcvd parameter of the final MC_RECEIVE_AND_WAIT is AP_DATA_COMPLETE_CONFIRM_DEALLOCATE, indicating that a confirmation of receipt of data is required before APPC will deallocate the conversation. The local TP waits for this confirmation until the partner TP issues MC_CONFIRMED.

Receiving Data Asynchronously

When using the Windows 2000, Windows NT, Windows 98, Windows 95, Windows version 3.*x*, and OS/2 operating systems, a TP can receive data asynchronously, without regard to other events occurring within the TP. The following table shows the methods by which a TP can receive data asynchronously. The table also indicates how asynchronous methods can be applied to actions other than receiving data.

Operating system	Method
Windows 2000, Windows NT, Windows 98, Windows 95, or Windows 3.*x*	**Through a Windows message:** Issue **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** with **WinAsyncAPPC**; the application is notified of completion through a **PostMessage** to the defined window handle. This method is not restricted to RECEIVE_AND_WAIT and MC_RECEIVE_AND_WAIT, but can be applied to any APPC verb.
Windows 2000, Windows NT, Windows 98, or Windows 95	**Through a Win32® event:** Issue **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** with **WinAsyncAPPCEx**; the application is notified of completion through a Win32 event. This method is not restricted to RECEIVE_AND_WAIT and MC_RECEIVE_AND_WAIT, but can be applied to any APPC verb.
Windows 2000, Windows NT, Windows 98, Windows 95, or OS/2	With **RECEIVE_AND_POST** or **MC_RECEIVE_AND_POST:** Issue the **RECEIVE_AND_POST** or **MC_RECEIVE_AND_POST** verb. With OS/2, the application is notified of completion through a semaphore. With Windows NT and Windows 95, the application is notified of completion through a Win32 event (the unsignaled event, not a semaphore, is passed in the **sema** member).

The following list gives details about these methods of receiving data asynchronously. For complete information, see the verb descriptions.

RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT with WinAsyncAPPC
: This method is defined as part of Windows SNA. It allows an application to issue a verb and be notified through a **PostMessage** when the action is complete. To retrieve the message number that will be posted to the window, call **RegisterWindowMessage** with "WinAsyncAPPC" as the input string. Then issue **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** using the **WinAsyncAPPC** entry point.

RECEIVE_AND_WAIT or **MC_RECEIVE_AND_WAIT** with WinAsyncAPPCEx
: This method allows an application to be notified through a Win32 event. This is particularly useful when writing applications that need to service multiple conversations simultaneously. The event must be in the nonsignaled state when passed to APPC, and the handle must have EVENT_MODIFY_STATE access to the event.

RECEIVE_AND_POST or MC_RECEIVE_AND_POST
: When using **RECEIVE_AND_POST** or **MC_RECEIVE_AND_POST** with Windows 2000, Windows NT, Window 98, or Windows 95, the application is notified through a Win32 event. The event must be in the nonsignaled state when passed to APPC, and the handle must have EVENT_MODIFY_STATE access to the event.

  When using RECEIVE_AND_POST or MC_RECEIVE_AND_POST with OS/2, the application is notified through a semaphore. Use DosSemSet to set the semaphore; APPC will clear the semaphore when the TP finishes receiving data asynchronously.

While receiving data asynchronously, the TP performs tasks not related to this conversation; the TP cannot issue most APPC verbs until notification is received. For information about the verbs that can be issued, see the descriptions of WinAsyncAPPC or WinAsyncAPPCEx.

After a verb has completed asynchronously, check the primary_rc to find out whether the data was received without error.

> If the initial call to issue the verb returns successfully, the application is guaranteed to be notified (by the applicable method) when the verb completes, regardless of whether the verb is ultimately successful.

Conversation Security

You can use conversation security to require that the invoking TP provide a user identifier and password before APPC will allocate a conversation with the invokable TP. If security is activated, the invoking TP must supply a combination of the user identifier and password as parameters of ALLOCATE or MC_ALLOCATE. Conversation security is activated and configured through registry or environment variables on the computer where the invokable TP is located.

With communication involving more than two TPs, the verification of a user identifier and password can be passed from one TP to another. Suppose that TP A invokes TP B, which requires security information, and TP B in turn invokes TP C, which also requires security information. Through ALLOCATE or MC_ALLOCATE, TP B can inform TP C that conversation security has already been verified.

For information about the registry or environment variables affecting conversation security, see Configuring Invokable TPs.

Basic and Mapped Conversations Compared

The following table offers some guidelines for choosing between basic and mapped conversations for your TPs. For definitions of basic and mapped conversations, see Fundamental Terms for TPs and LUs.

Characteristic	Basic conversations	Mapped conversations
Common use	Generally used for service TPs.	Generally used for application TPs.
Partnering	Must be used to communicate with an existing TP that uses basic verbs.	Must be used to communicate with an existing TP that uses mapped verbs.
Sending and receiving method	Before a TP can begin a send operation, it must convert data records into logical records. The TP does this by adding a 2-byte prefix that indicates the length of the record. A TP can send several logical records at one time. When a partner TP receives logical records, it must reconstruct them into usable data records. For more information, see Logical Records Used in Basic Conversations.	A TP sends data one record at a time. Neither the sending TP nor the receiving TP needs to convert data records between different forms.
Abnormal termination	In the DEALLOCATE verb, a TP can indicate whether an error or ABEND (abnormal program termination) was caused by a TP or by a program using the TP.	A TP can indicate an error or ABEND, but cannot tell whether a problem was caused by a TP or by a program using a TP.
	A TP can indicate whether an ABEND was caused by a timeout or by a critical error.	A TP cannot indicate the cause of an ABEND.
Error logging	For an error or ABEND, a TP can send an error message, in the form of a general data stream (GDS) error log variable, to the local log and to the partner LU.	For an error or ABEND, a TP cannot send an error message to the local log or to the partner LU.

This section contains:

- Logical Records Used in Basic Conversations
- An Example of a Mapped Conversation

Logical Records Used in Basic Conversations

Logical records are sent and received in basic conversations only.

A TP can send or receive multiple logical records with a single SEND_DATA or receive verb. The receive verbs are RECEIVE_AND_POST (Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2), RECEIVE_IMMEDIATE, and RECEIVE_AND_WAIT. A TP can also send or receive a logical record in successive portions: beginning, middle, and end.

A logical record is made up of:

- A 2-byte record-length (LL) field.
- A data field that can range in length from 0 bytes through 32765 bytes.

The LL field contains a hexadecimal value that is the length of the data field plus two bytes (for the LL field). For example, if a record contains 228 bytes of application data, the logical record length is 230. The LL field is 0x00E6, the hexadecimal equivalent of 230. If the length of the data field is 0, the value contained in the LL field is 0x0002.

Logical records are sent from or received in a data buffer established by the TP. In the data buffer, the LL field must not be in Intel byte-swapped format. For example, a length of 230 must be 0x00E6, not 0xE600.

The LL field cannot be 0x0000 or 0x0001, which allow less than the two bytes required for the LL field itself. The LL field also cannot be greater than or equal to 0x8000, which is equivalent to decimal 32768 and therefore allows for a data field greater than 32765 or an LL field greater than 2.

Setting the most significant bit of the LL field to 1 indicates that the information contained in the current logical record is continued in the next logical record.

An Example of a Mapped Conversation

For background information about mapped conversations, see Basic and Mapped Conversations Compared.

The following example of a mapped conversation shows the APPC verbs used to start a conversation, exchange data, and end the conversation. APPC verb parameters are in parentheses.

Issued by the invoking TP	Issued by the invokable TP
TP_STARTED	
MC_ALLOCATE	
MC_SEND_DATA	
MC_DEALLOCATE	
TP_ENDED	RECEIVE_ALLOCATE
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_OK)
	(rtn_status=AP_NO)
	(what_rcvd=AP_DATA_COMPLETE)
	MC_RECEIVE_AND_WAIT
	(primary_rc=AP_DEALLOC_NORM)
	TP_ENDED

The following paragraphs describe the verbs that are used in a mapped conversation.

Verbs for Starting a Mapped Conversation

To start a mapped conversation, the invoking TP issues the following verbs:

- TP_STARTED, which notifies APPC that the local TP is beginning a conversation.
- MC_ALLOCATE, which requests that APPC establish a conversation between the local TP and the partner TP.

The invokable TP issues RECEIVE_ALLOCATE, which informs APPC that it is ready to begin a conversation with the invoking TP.

Verbs for Sending Data in a Mapped Conversation

MC_SEND_DATA puts one data record (a record containing application data to be transmitted) in the send buffer of the local LU. Data transmission to the partner TP does not happen until one of the following events occurs:

- The send buffer fills up.
- The sending TP issues a verb that forces APPC to flush the buffer and send data to the partner TP.

In the preceding example, the send buffer contains both the data record and the MC_ALLOCATE request (which precedes the data record). Therefore, in the example, MC_DEALLOCATE flushes the buffer, sending the **MC_ALLOCATE** request and data record to the partner TP. Other verbs that flush the buffer are MC_CONFIRM and MC_FLUSH.

Verbs for Receiving Data in a Mapped Conversation

The MC_RECEIVE_AND_WAIT verb allows a TP to receive a data record or status information. If no data is currently available, the TP waits for data to arrive. For Windows 2000, Windows NT, Windows 98, Windows 95, and Windows 3.*x* systems, issue **MC_RECEIVE_AND_WAIT** in conjunction with WinAsyncAPPC rather than the blocking version of this call.

In the example, the receiving TP issues MC_RECEIVE_AND_WAIT twice. The first time, it issues the verb to receive data. When it finishes receiving the complete data record (what_rcvd is AP_DATA_COMPLETE), it issues MC_RECEIVE_AND_WAIT again to receive a return code. The return code AP_DEALLOC_NORMAL indicates that the conversation has been deallocated.

> MC_RECEIVE_IMMEDIATE performs the same function as **MC_RECEIVE_AND_WAIT**, except that it does not wait if data is not currently available from the partner TP. Instead, it returns a no-data-available response to the calling TP.

Verbs for Ending a Mapped Conversation

To end a mapped conversation, one of the TPs issues MC_DEALLOCATE, which causes APPC to deallocate the conversation between the two TPs.

After the conversation has been deallocated, both TPs issue TP_ENDED.

> A TP can participate in multiple conversations simultaneously. In this case, the TP issues **TP_ENDED** after all conversations have been deallocated.

Using Invoking and Invokable TPs

There are two kinds of TPs: TPs that can invoke (that is, initiate a conversation with) other TPs, and TPs that can be invoked. A TP that can invoke another TP is called an invoking TP, and a TP that can be invoked is called an invokable TP.

The following topics describe how:

- Invoking TPs request invokable TPs.
- Invokable TPs identify themselves to Host Integration Server or SNA Server in preparation for being invoked.
- An invokable TP is matched to an invoking TP's request.

For information about how to configure LUs to support TPs, see Configuring TPs on Host Integration Server and SNA Server and the *Microsoft Host Integration Server 2000 online books*.

This section contains:

Invoking TPs

An invoking TP can be located on any system on the SNA network. An invoking TP identifies itself by issuing TP_STARTED, which specifies the name of the invoking TP and can specify the LU alias that the TP uses. If the LU alias is not specified in **TP_STARTED**, Host Integration Server or SNA Server must be configured to supply it through one of two types of default local LU; otherwise, **TP_STARTED** will fail. For more information, see Configuring Invoking TPs on Host Integration Server and SNA Server.

Next, the invoking TP initiates the invoking process by issuing ALLOCATE or MC_ALLOCATE, in which it specifies the name of the invokable TP, and can also specify the partner LU alias (the LU alias to be used by the invokable TP). If the partner LU is not specified in ALLOCATE or MC_ALLOCATE, Host Integration Server 2000 must be configured to supply one through the default remote APPC LU assigned to the user who started the invoking TP; otherwise, ALLOCATE or MC_ALLOCATE will fail. For more information, see Configuring Invoking TPs on Host Integration Server and SNA Server.

After a TP successfully issues an ALLOCATE or MC_ALLOCATE verb, an allocation request flows. For more information about what happens after an invoking TP requests an invokable TP, see Matching Invoking and Invokable TPs.

Invoking TPs and Contention

The following information applies only to cases where LUs are communicating in complex ways (such as chains of LUs) over multiple sessions. In such cases, two LUs may attempt to allocate a conversation on the same session at the same time. If this happens, one LU must win (the contention winner) and one must lose (the contention loser). The contention-winner LU and the contention-loser LU are determined for each session when the session is established. During that particular session, the contention-loser LU must receive permission from the contention-winner LU before allocating a conversation. In contrast, the contention-winner LU on that session allocates a conversation as needed.

Note that when two LUs are communicating over multiple sessions, one LU can be the contention winner for some of the sessions, and the other LU the contention winner for others.

An invoking TP will operate most efficiently if the number of concurrent ALLOCATE or MC_ALLOCATE requests that the TP issues is matched by the number of sessions on which the local LU is the contention winner. The choice of contention winner is controlled through the modes configured at the two ends of the communication. The mode is configured using SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0. A mode must be configured to work with the mode on the remote system for communication to begin between two LUs. For more information about modes, see the *Microsoft Host Integration Server 2000 online books*.

Invokable TPs

An invokable TP is a TP that can be invoked by another TP. Invokable TPs are written or configured through registry or environment variables to supply their names to Host Integration Server 2000 or SNA Server as a notification that they are available for incoming requests. Invokable TPs can be run on any Host Integration Server 2000 or SNA Server client or server running Windows 2000, Windows NT, Windows 98, Windows 95, Windows 3.*x*, or OS/2.

> Invokable TPs cannot be run on Microsoft MS-DOS®-based clients.

There are two types of invokable TPs:

Operator-started invokable TPs

An operator-started invokable TP must be started by an operator before the TP can be invoked. When the operator-started invokable TP is started, it notifies Host Integration Server 2000 or SNA Server of its availability by issuing a RECEIVE_ALLOCATE verb. The **RECEIVE_ALLOCATE** causes the name of the invokable TP along with the alias of an associated LU if one has been configured through a registry or environment variable to be communicated to all the servers running Host Integration Server 2000 or SNA Server in the SNA domain.

Autostarted invokable TPs

An autostarted invokable TP can be started by Host Integration Server 2000 or SNA Server when needed. The TP must be registered through registry entries or environment variables on its local system, so that it can be identified to the SnaBase component of the Host Integration Server 2000 or SNA Server client software. The registered information defines the TP as autostarted and must specify the TP name. The registered information can also specify the local LU alias that the invokable TP will use.

The recommended method for setting registry or environment variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP, or similar code written into your own installation program. For more information about registry or environment variables for invokable TPs, see Configuring Invokable TPs. For information about TPSETUP, see Sample APPC TPs in the SDK.

If no local LU alias is registered with autostarted TPs, the resulting Host Integration Server 2000 or SNA Server configuration can be more flexible in responding to invoking requests. For more information about such flexible configurations, see TP Name Not Unique; Local LU Alias Unspecified.

After an autostarted invokable TP is started by Host Integration Server 2000, the TP issues RECEIVE_ALLOCATE just as an operator-started TP does. RECEIVE_ALLOCATE must provide the TP name that was registered for the TP.

Autostarted TPs must be configured through registry or environment variables to be either queued or nonqueued. All operator-started TPs act as queued TPs.

Queued TPs

If an autostarted TP is configured as queued, or if the TP is operator-started, incoming allocation requests are queued and then sent only when the invokable TP issues **RECEIVE_ALLOCATE**. For autostarted invokable TPs, if a copy of the TP is not yet running, one is started when an incoming allocation request specifies that TP.

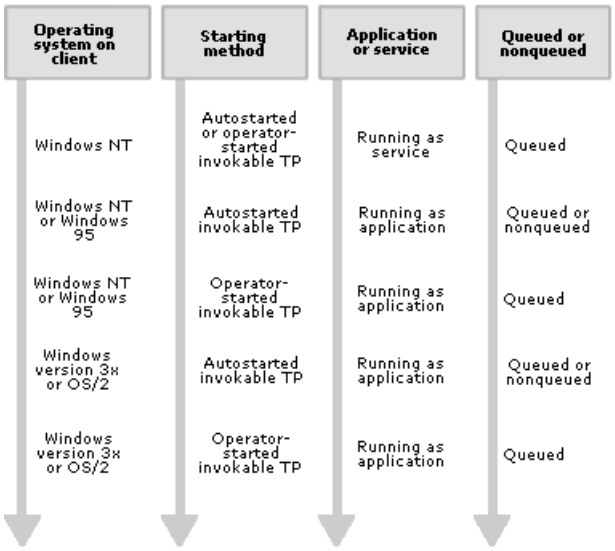
> For Windows 2000 and Windows NT, only one copy of a service can be running at any given time; this means that all autostarted TPs that run as services under Windows 2000 or Windows NT must be queued. To write an autostarted TP so it will run under Windows 2000 or Windows NT as a service and also run in a nonqueued way, write a multithreaded program with a **RECEIVE_ALLOCATE** always outstanding.

Nonqueued TPs

If an autostarted TP is configured as nonqueued, a new copy will be started every time an ALLOCATE or MC_ALLOCATE is received for the TP. Nonqueued TPs should process the conversation they have been allocated and then exit, since they will not receive any additional **ALLOCATE** or **MC_ALLOCATE** requests.

Subcategories for Invokable TPs

The following figure shows subcategories for invokable TPs.



The concept of a TP "running as a service" or "running as an application" is distinct from a service TP or an application TP. Service TP and application TP are SNA terms that describe how a TP is used: either as a supportive service program for other APPC programs, or directly by a user, as an application. For detailed information about services and applications on Windows 2000 and Windows NT, see the Microsoft Developer Network (MSDN®) Platform Software Development Kit.

To write an autostarted TP so it will run under Windows 2000 or Windows NT as a service and also run in a nonqueued way, write a multithreaded program with a RECEIVE_ALLOCATE always outstanding. See Invokable TPs.

To run an autostarted TP as an application under Windows NT, make sure the TPSTART program is always started before the TP. See Sample APPC TPs in the SDK.

Matching Invoking and Invokable TPs

Each computer running Host Integration Server 2000 or SNA Server maintains a list of available invokable TP names and any LU aliases to be associated with the TP names. This information is obtained as follows:

- For autostarted invokable TPs, registry or environment variables identify a TP name containing a maximum of eight characters, and can specify an associated LU. This information is sent from the client to the server that sponsors the client. A client learns about the domain through a sponsor connection to a server; clients must establish the sponsor connection before proceeding with any other tasks.
- For operator-started invokable TPs, a TP name (with a maximum of 64 characters) is specified with the RECEIVE_ALLOCATE verb. The TP name is truncated to eight characters and sent from the client to the server that sponsors the client, along with the alias of an associated LU if one has been configured through a registry or environment variable.

> If you want a TP name to be unique, it is recommended that you limit the name to eight characters or fewer, or make the name unique within the first eight characters. This is because the preliminary routing of allocation requests is carried out using the first eight characters. Although further matching is later carried out between the full TP names specified in ALLOCATE or MC_ALLOCATE and **RECEIVE_ALLOCATE**, it is inefficient to allow the preliminary routing to succeed when in some cases the later matching will fail.

The next step in the matching of invoking and invokable TPs is that the invoking TP issues the **ALLOCATE** or **MC_ALLOCATE** verb. After an invoking TP in an Host Integration Server or SNA Server domain successfully issues this verb, an allocation request flows to the partner LU specified in the **ALLOCATE** or **MC_ALLOCATE** verb, stating the name of the invokable TP that has been requested.

When an allocation request arrives, the SNA server compares the requested invokable TP name and LU alias to the list of available invokable TPs (which can include associated LU aliases). The comparison can be modified by registry variables, but by default is carried out as follows:

- Although the TP name requested in the **ALLOCATE** or **MC_ALLOCATE** verb can be as long as 64 characters, any name received through a registry or environment variable is limited to eight characters or less. Therefore, only the first eight characters of TP names are used in comparisons.
- The comparison is carried out first on both the TP name and the LU alias. An invokable TP for which there is a match on both TP name and LU alias will be chosen ahead of a TP for which no LU alias has been configured through a registry or environment variable. A TP for which no LU alias has been configured can be matched with any request that specifies that TP name, since there cannot be a mismatch based on LU alias.
- The comparison of requested and available TP names is carried out in a specific order:

  1. Host Integration Server or SNA Server first checks for operator-started invokable TPs on the local system (the local computer running Host Integration Server 2000 or SNA Server).
  2. If no match is found, Host Integration Server or SNA Server checks for autostarted invokable TPs on the local system (the local computer running Host Integration Server 2000 or SNA Server).
  3. If no match is found, Host Integration Server or SNA Server checks for operator-started invokable TPs on other Host Integration Server 2000 or SNA Server clients or servers.
  4. If no match is found, Host Integration Server or SNA Server checks for autostarted invokable TPs on other Host Integration Server or SNA Server clients or servers.

This comparison can be modified somewhat by registry entries for the SnaServr service. The entries are called **DloadMatchTPOnly** and **DloadMatchLocalFirst**, and are described in the *Microsoft Host Integration Server 2000 Reference online book*.

If a match is found, the Host Integration Server or SNA Server signals the system containing the requested TP to connect to that server running Host Integration Server 2000 or SNA Server. If no match is found, Host Integration Server or SNA Server rejects the incoming request.

For suggestions about specific ways to handle TP names and LU aliases, see Arranging TPs Within an SNA Network.

> Because of the way APPC works, an allocation request will not flow until local data buffers are full, or a confirm or flush verb is issued. This can mean that the allocation request does not flow until some time after the ALLOCATE or MC_ALLOCATE verb is issued. Therefore, any allocation failure caused by the rejection of the allocation request at the partner LU will be observed as the failure of a later verb with one of the allocation failure return codes.

Configuring Invokable TPs

The following topics discuss how to configure invokable TPs for the various Microsoft Host Integration Server 2000 or Microsoft SNA Server client types.

This section contains:

Clients Running Windows 2000 or Windows NT

On clients running Windows 2000 or Windows NT, invokable TPs are configured through the Windows 2000 or Windows NT registry.

> With Windows 2000 or Windows NT, the recommended method for setting registry variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile INSTALL.C, the source code for TPSETUP, for the Windows NT environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample APPC TPs in the SDK.

For clients running Windows 2000 or Windows NT, it is recommended that autostarted invokable TPs be written as Windows 2000 or Windows NT services. Be sure to include code like that in TPSETUP in the program that installs your TPs. Among other things, TPSETUP shows how to use the **CreateService** function when installing a TP. For important information about how services work under Windows 2000 and Windows NT, see the Microsoft Developer Network (MSDN) Platform Software Development Kit for Windows 2000 and Windows NT.

The following table lists the registry entries used for the types of invokable TPs that can be run on Windows-2000 or Windows NT clients:

Type of TP	**Location in registry**	**Possible registry entries**
Autostarted invokable TP running as a service on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services *TPName*** (and subkeys)	Registry entries created by the **CreateService** call, including entries that specify the path, display name, and other characteristics of the service. —plus— Linkage OtherDependencies:REG_MULTI_SZ:SnaBase Parameters SNAServiceType:REG_DWORD:0x5 LocalLU:REG_SZ:*LUalias* Parameters:REG_SZ:*ParameterList* Timeout:REG_DWORD:*number* ConversationSecurity:REG_SZ:{ YES | NO } AlreadyVerified:REG_SZ:{ YES | NO }[2] *Username1*:REG_SZ:*Password1*[2] ... *UsernameX*:REG_SZ:*PasswordX*[2]
Autostarted invokable TP running as an application1 on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters TPs *TPName*** Parameters	**SNAServiceType:**REG_DWORD:{ 0x5 | 0x6 } **PathName:**REG_EXPAND_SZ:*path* **LocalLU:**REG_SZ:*LUalias* **Parameters:**REG_SZ:*ParameterList* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }2 ***Username1*:**REG_SZ:*Password1*[2] ... ***UsernameX*:**REG_SZ:*PasswordX*[2]

Operator-started invokable TP running as a service on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services *TPName*** (and subkeys)	Registry entries created by the **CreateService** call, including entries that specify the path, display name, and other characteristics of the service. —plus— Linkage OtherDependencies:REG_MULTI_SZ:SnaBase Parameters SNAServiceType:REG_DWORD:0x1A LocalLU:REG_SZ:*LUalias* Timeout:REG_DWORD:*number* ConversationSecurity:REG_SZ:{ YES | NO } AlreadyVerified:REG_SZ:{ YES | NO }[2] *Username1*:REG_SZ:*Password1*[2] ... *UsernameX*:REG_SZ:*PasswordX*[2]
Operator-started invokable TP on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters TPs *TPName*** Parameters	**SNAServiceType:**REG_DWORD:0x1A **LocalLU:**REG_SZ:*LUalias* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }2 ***Username1:***REG_SZ:*Password1*[2] ... ***UsernameX:***REG_SZ:*PasswordX*[2]

Notes

1 Before an autostarted TP can be started as an application on a Windows 2000 or Windows NT client, the TPSTART program must be started. For more information, see Sample APPC TPs in the SDK.

2 AlreadyVerified and Username/Password entries are used only if ConversationSecurity is set to YES.

This section contains:

- Registry Entries for Clients Running Windows 2000 or Windows NT
- Example of Windows 2000 or Windows NT Registry Entries for an Invokable TP

Registry Entries for Clients Running Windows 2000 or Windows NT

The following list gives details about registry entries for clients running Windows 2000 or Windows NT. For each TP type, the applicable variables and their locations are shown in Clients Running Windows 2000 or Windows NT.

Registry Entries for TPName on Clients Running Windows 2000 or Windows NT

TPName:REG_MULTI_SZ
The name of the transaction program (TP) that is executed. A TP name is up to 64 ASCII characters in length and cannot contain spaces or nulls.

SNA service TPs are a special set of TPs defined by the SNA protocols. Each service TP is a specially-defined function with a special name. An SNA service TP name is represented by up to four EBCDIC bytes; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII TP name form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for WIN.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, an EBCDIC service TP name of 0x21 0xD7 0xD7 is equivalent to a TP name of SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Registry Entries for the TPName Subtree on Clients Running Windows 2000 or Windows NT

OtherDependencies:REG_MULTI_SZ:SnaBase
For a TP running as a service, ensures that the SnaBase service will be started before the TP is started. This entry belongs under the **Linkage** subkey.
SNAServiceType:REG_DWORD:{ 0x5 | 0x6 | 0x1A }
Indicates the type of TP. Use a value of 0x5 for an autostarted queued TP, 0x6 for an autostarted nonqueued TP, and 0x1A for an operator-started TP.

Note that the value for an autostarted TP running as a service must be 0x5, because these TPs are always queued, as described in Invokable TPs.

PathName:REG_SZ**:***path*
For an autostarted TP running as an application, specifies the path and file name of the TP. The data type of REG_EXPAND_SZ means that the path can contain an expandable data string; for example, %SystemRoot% represents the directory containing the Windows NT system files. Note that for a TP running as a service, an equivalent entry is inserted by the **CreateService** call; no additional path entry is needed.
LocalLU:REG_SZ:*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.
Parameters:REG_SZ:*ParameterList*
Lists parameters to be used by the TP. Separate parameters with spaces.
Timeout:REG_DWORD:*number*
Specifies the time, in milliseconds, that a RECEIVE_ALLOCATE will wait before timing out. Specify *number* in decimal; the registry editor converts this to hexadecimal before displaying it. The default is infinity (no limit).
ConversationSecurity:REG_SZ:{ YES | NO }

Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified:REG_SZ:{ YES | NO }

Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO. The default value is NO.

For a diagram of three TPs in a conversation, where the third TP can be invoked with a password that is already verified by the second TP, see Communication Between TPs. The following table shows the requirements for using password verification in a chain of TPs.

First TP (invoking TP)	**Second TP (invokable TP that confirms password and then invokes another TP)**	**Third and subsequent TPs (invokable TPs that invoke other TPs)**
Does not need registry or environment variables.	**ConversationSecurity** setting must be YES.	**ConversationSecurity** setting must be YES.
Does not need registry or environment variables.	**AlreadyVerified** setting can be YES or NO.	**AlreadyVerified** setting must be YES.
ALLOCATE or MC_ALLOCATE in this TP has a **security** parameter of AP_PGM; as a result, the TP passes along the **user_id** and **pwd** values supplied in **ALLOCATE** or **MC_ALLOCATE**.	**ALLOCATE** or **MC_ALLOCATE** in this TP has a **security** parameter of AP_SAME; as a result, after confirming the user identifier and password, the TP passes along the user identifier and an already-verified flag.	**ALLOCATE** or **MC_ALLOCATE** in this TP has a **security** parameter of AP_SAME; as a result, the TP passes along the user identifier as received, along with the already-verified flag.

If you set **AlreadyVerified** to NO, this TP cannot join in a chain of conversations where password verification is already done. The exception to this is when **ConversationSecurity** is set to NO, in which case the TP could be the final TP in such a chain, since it performs no checking.

If you are configuring a TP that sometimes needs to confirm a password and sometimes accepts an already-verified flag, set AlreadyVerified to YES and configure the *UsernameX* variable appropriately. In this case, whenever the TP is invoked without the already-verified flag set, AlreadyVerified is ignored; verification is attempted with the user identifier and password configured for the TP.

If you want to have a chain of conversations where the user identifier and password are reverified at every step, carry out the following. For all the TPs, set ConversationSecurity to YES, and in each ALLOCATE or MC_ALLOCATE issued, set the security parameter to AP_PGM and the pwd and user_id parameters to valid combinations.

If you set AlreadyVerified to YES, make sure that ConversationSecurity is also set to YES.

Username1:REG_SZ:Password1

...

UsernameX:REG_SZ:PasswordX

Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can each be as many as 10 characters. Both parameters are case-sensitive.

This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the AlreadyVerified entry.

Example of Windows 2000 or Windows NT Registry Entries for an Invokable TP

For an autostarted invokable TP called **BounceTP** and running as a service, the following registry entries might be added to a Windows 2000 or Windows NT-based client. The entries would be added to **HKEY_LOCAL_MACHINE\ SYSTEM\ CurrentControlSet\ Services\ SnaBase**, under the subkeys shown in bold type.

> In the following list, the parameters listed directly under the **BounceTP** key (such as **DisplayName** and **ErrorControl**) are service parameters created when TPSETUP or similar code is run to install the TP. These parameters should be created by TPSETUP or similar code; they should not be set manually. For more information about TPSETUP, see Sample APPC TPs in the SDK.

BounceTP
DisplayName:REG_SZ:BounceTP
ErrorControl:REG_DWORD:0x1
ImagePath:REG_EXPAND_SZ:c:\sna\system\bouncetp.exe
ObjectName:REG_SZ:LocalSystem
Start:REG_DWORD:0x3
Type:REG_DWORD:0x10

Linkage

OtherDependencies:REG_MULTI_SZ:SnaBase

Parameters

SNAServiceType:REG_DWORD:0x5
LocalLU:REG_SZ:JohnDoe
Parameters:REG_SZ:Arg1 Arg2 Arg3
Timeout:REG_DWORD:0x100
ConversationSecurity:REG_SZ:yes
AlreadyVerified:REG_SZ:no
JohnDoe:REG_SZ:SecretPassword

Security

Security:REG_BINARY:

For an autostarted invokable TP called **BounceTP** running as an application, the following registry entries might be added to a Windows NT-based client. The entries would be added to **HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\Services\SnaBase\Parameters\Tps**, under the subkeys shown in bold type.

> In the following list, the parameters listed under the **BounceTP** key (such as **PathName** and **ConversationSecurity**) are parameters created when TPSETUP or similar code is run to install the TP. These parameters should be created by TPSETUP or similar code; they should not be set manually. For more information about TPSETUP, see Sample APPC TPs in the SDK.

BounceTP
Parameters

SNAServiceType:REG_DWORD:0x5
PathName:REG_SZ:C:\sna\system\bouncetp.exe
LocalLU:REG_SZ:JohnDoe
Parameters:REG_SZ:Arg1 Arg2 Arg3
Timeout:REG_DWORD:0x100
ConversationSecurity:REG_SZ:yes
AlreadyVerified:REG_SZ:no
JohnDoe:REG_SZ:SecretPassword

Clients Running Windows 98 or Windows 95

On clients running Windows 98 or Windows 95, invokable TPs are configured through the Windows 98 or Windows 95 registry.

> On Windows 98 or Windows 95, the recommended method for setting registry variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile the source code for TPSETUP (INSTALL.C,), for the Windows 98 or Windows 95 environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample APPC TPs in the SDK.

The following table lists the registry entries used for the types of invokable TPs that can be run on Windows 98 or Windows 95 clients:

Type of TP	**Location in registry**	**Possible registry entries**
Autostarted invokable TP running as an application on a Windows 98 or Windows 95 client	**HKEY_LOCAL_MACHINE** **SYSTEM** **SOFTWARE** **Microsoft** **SnaBase** **Parameters** **TPs** ***TPName*** Parameters	**SNAServiceType:**REG_DWORD:{ 0x5 | 0x6 } **PathName:**REG_EXPAND_SZ:*path* **LocalLU:**REG_SZ:*LUalias* **Parameters:**REG_SZ:*ParameterList* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO } ***Username1*:**REG_SZ:*Password1* ... ***UsernameX*:**REG_SZ:*PasswordX*
Operator-started invokable TP on a Windows 98 or Windows 95 client	**HKEY_LOCAL_MACHINE** **SYSTEM** **SOFTWARE** **Microsoft** **SnaBase** **Parameters** **TPs** ***TPName*** Parameters	**SNAServiceType:**REG_DWORD:0x1A **LocalLU:**REG_SZ:*LUalias* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO } ***Username1*:**REG_SZ:*Password1* ... ***UsernameX*:**REG_SZ:*PasswordX*[1]

Note

1 **AlredyVerified** and Username/Password entries are used only if **ConversationSecurity** is set to YES.

This section contains:

- Registry Entries for Clients Running Windows 98 or Windows 95
- Example of Windows 98 or Windows 95 Registry Entries for an Invokable TP

Registry Entries for Clients Running Windows 98 or Windows 95

The following list gives details about registry entries for clients running Windows 98 or Windows 95. For each TP type, the applicable variables and their locations are shown in Clients Running Windows 98 or Windows 95.

Registry Entries for TPName on Clients Running Windows 98 or Windows 95

TPName:REG_MULTI_SZ
The name of the transaction program (TP) that is executed. A TP name is up to 64 ASCII characters in length and cannot contain spaces or nulls.

SNA service TPs are a special set of TPs defined by the SNA protocols. Each service TP is a specially-defined function with a special name. An SNA service TP name is represented by up to four EBCDIC bytes; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII TP name form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for WIN.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, an EBCDIC service TP name of 0x21 0xD7 0xD7 is equivalent to a TP name of SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Registry Entries for the TPName Subtree on Clients Running Windows 95

SNAServiceType:REG_DWORD:{ 0x5 | 0x6 | 0x1A }
Indicates the type of TP. Use a value of 0x5 for an autostarted queued TP, 0x6 for an autostarted nonqueued TP, and 0x1A for an operator-started TP.

PathName:REG_SZ:*path*
For an autostarted TP running as an application, specifies the path and file name of the TP. The data type of REG_EXPAND_SZ means that the path can contain an expandable data string; for example, %SystemRoot% represents the directory containing the Windows 95 system files.

LocalLU:REG_SZ:*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters:REG_SZ:*ParameterList*
Lists parameters to be used by the TP. Separate parameters with spaces.

Timeout:REG_DWORD:*number*
Specifies the time, in milliseconds, that a RECEIVE_ALLOCATE will wait before timing out. Specify *number* in decimal; the registry editor converts this to hexadecimal before displaying it. The default is infinity (no limit).

ConversationSecurity:REG_SZ:{ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified:REG_SZ:{ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO. The default value is NO.

For a diagram of three TPs in a conversation, where the third TP can be invoked with a password that is already verified by the second TP, see Communication Between TPs. The following table shows the requirements for using password verification in a chain of TPs.

First TP (invoking TP)	Second TP (invokable TP that confirms password and then invokes another TP)	Third and subsequent TPs (invokable TPs that invoke other TPs)
Does not need registry or environment variables.	**ConversationSecurity** setting must be YES.	**ConversationSecurity** setting must be YES.
Does not need registry or environment variables.	**AlreadyVerified** setting can be YES or NO.	**AlreadyVerified** setting must be YES.
ALLOCATE or MC_ALLOCATE in this TP has a **security** parameter of AP_PGM; as a result, the TP passes along the **user_id** and **pwd** values supplied in **ALLOCATE** or **MC_ALLOCATE**.	**ALLOCATE** or **MC_ALLOCATE** in this TP has a **security** parameter of AP_SAME; as a result, after confirming the user identifier and password, the TP passes along the user identifier and an already-verified flag.	**ALLOCATE** or **MC_ALLOCATE** in this TP has a **security** parameter of AP_SAME; as a result, the TP passes along the user identifier as received, along with the already-verified flag.

If you set **AlreadyVerified** to NO, this TP cannot join in a chain of conversations where password verification is already done. The exception to this is when **ConversationSecurity** is set to NO, in which case the TP could be the final TP in such a chain, since it performs no checking.

If you are configuring a TP that sometimes needs to confirm a password and sometimes accepts an already-verified flag, set AlreadyVerified to YES and configure the *UsernameX* variable appropriately. In this case, whenever the TP is invoked without the already-verified flag set, AlreadyVerified is ignored; verification is attempted with the user identifier and password configured for the TP.

If you want to have a chain of conversations where the user identifier and password are reverified at every step, carry out the following. For all the TPs, set ConversationSecurity to YES, and in each ALLOCATE or MC_ALLOCATE issued, set the security parameter to AP_PGM and the pwd and user_id parameters to valid combinations.

If you set AlreadyVerified to YES, make sure that ConversationSecurity is also set to YES.

Username1:REG_SZ:Password1

...

UsernameX:REG_SZ:PasswordX

Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can each be as many as 10 characters. Both parameters are case-sensitive.

This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the AlreadyVerified entry.

Example of Windows 98 or Windows 95 Registry Entries for an Invokable TP

For an autostarted invokable TP called **BounceTP** and running as an application, the following registry entries might be added to a Windows 98 or Windows 95 client. The entries would be added to **HKEY_LOCAL_MACHINE\ SYSTEM\ SOFTWARE\ Microsoft\ SnaBase\ Parameters\ TPs**, under the subkeys shown in bold type.

> In the following list, the parameters listed directly under the **BounceTP Parameters** key (such as **DPathName** and **AlreadyVerified**) are parameters created when TPSETUP or similar code is run to install the TP. These parameters should be created by TPSETUP or similar code; they should not be set manually. For more information about TPSETUP, see Sample APPC TPs in the SDK.

BounceTP
Parameters

SNAServiceType:REG_DWORD:0x5
PathName:REG_SZ:C:\sna\system\bouncetp.exe
LocalLU:REG_SZ:JohnDoe
Parameters:REG_SZ:Arg1 Arg2 Arg3
Timeout:REG_DWORD:0x100
ConversationSecurity:REG_SZ:yes
AlreadyVerified:REG_SZ:no
JohnDoe:REG_SZ:SecretPassword

Clients Running Windows Version 3.x

On clients running Windows version 3.*x*, invokable TPs are configured through entries in the WIN.INI file.

> With Windows version 3.*x*, the recommended method for setting environment variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile the source code for TPSETUP (INSTALL.C) for the Windows version 3.*x* environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample APPC TPs in the SDK.

The following table lists the section headings and environment variables used in the WIN.IN file for invokable TPs on clients running Windows version 3.*x*:

Type of TP	**Section in WIN.INI listing TP names only**	**Section and possible environment variables defining TP**
Autostarted invokable TP on a client running Windows version 3.*x*	**[SNAServerAutoTPs]** ***TPName1***=*SectionName1* ... ***TPNameX***=*SectionNameX*	**[*SectionName1*]** **PathName**=*path* **LocalLU**=*LUalias* **Parameters**=*ParameterList* **TimeOut**=*number* **Queued**={ YES | NO } **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }1 ***Username1***=*Password1*1 ... ***UsernameX***=*PasswordX*1
Operator-started invokable TP on a client running Windows version 3.*x*	**[SNAServerAutoTPs]** ***TPNameN***=*SectionNameN* ... ***TPNameX***=*SectionNameX*	**[*SectionNameN*]** **LocalLU**=*LUalias* **TimeOut**=*number* **Queued**=OPERATOR **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }1 ***Username1***=*Password1*1 ... ***UsernameX***=*PasswordX*1

Note

1 **AlreadyVerified** and Username/Password lines are used only if **ConversationSecurity** is set to YES.

This section contains:

- Environment Variables for Clients Running Windows Version 3.x
- Translating SNA Service TP Names to ASCII for WIN.INI
- Example of WIN.INI Lines for an Invokable TP

Environment Variables for Clients Running Windows Version 3.x

The following list shows the correct form for the sections and entries to add to the WIN.INI file for autostarted invokable TPs on a client running Windows version 3.*x*. The section headings are shown enclosed in square brackets; include the brackets when adding the section to the WIN.INI file.

For each TP type, the applicable variables and their locations are shown in Clients Running Windows Version 3.x.

[SNAServerAutoTPs]
TPNameX=*SectionNameX*
For all TPs. Associates *TPnameX* with *SectionNameX*. Additional lines can follow *TPNameX*=*SectionNameX*, each one using the same syntax to name a different TP and the section containing the information for that TP.
[*SectionName*]
Forms a section heading for entries applying to one TP. *SectionName* must match a section name listed under **[SNAServerAutoTPs]**.
PathName=*path*
Specifies the full path and file name of the executable file. The default is TPNAME.EXE.
LocalLU=*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.
Parameters=*ParameterList*
Lists strings to be passed as command-line parameters for the TP. Separate parameters with spaces. The default is no parameters.
Timeout=*number*
Specifies the time in milliseconds that a RECEIVE_ALLOCATE will wait before timing out. The default is infinity (no limit).
Queued={ YES | NO }
Queued=OPERATOR
Specifies the type of TP: YES for an autostarted queued TP, NO for an autostarted nonqueued TP, or OPERATOR for an operator-started TP (which must always be queued). The default is YES.
ConversationSecurity={ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.
AlreadyVerified={ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For detailed information about the **AlreadyVerified** variable, see its description under Registry Entries for Clients Running Windows 2000 or Windows NT.

The default is NO.

Username1=Password1
...
UsernameX=PasswordX
Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can have as many as 10 characters each. Both parameters are case-sensitive. This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the **AlreadyVerified** entry.

Translating SNA Service TP Names to ASCII for WIN.INI

For SNA service TPs on Host Integration Server 2000 or SNA Server clients running Windows version 3.*x*, the line naming the TP in the WIN.INI file must specify the TP name in ASCII. The following paragraphs tell how to convert a TP name to this form. The line should be placed in the **[SNAServerAutoTPs]** section of the file, as shown in Clients Running Windows Version 3.x.

An SNA service TP name is normally up to four bytes in length; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for WIN.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, a service TP name of 0x21 0xD7 0xD7 is equivalent to SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Example of WIN.INI Lines for an Invokable TP

For autostarted invokable TPs called **BounceTP** and **TestTP** on a client running Windows version 3.*x*, the following WIN.INI lines might be added:

```
[SNAServerAutoTPs]
BounceTP=bnceprms
TestTP=testprms

[bnceprms]
PathName=c:\sna\wbounce.exe
LocalLU=Eric
Parameters=/t
timeout=60000
queued=yes

[testprms]
PathName=c:\sna\testtp.exe
LocalLU=LU1
Parameters=/v
timeout=60000
queued=no
```

Clients Running OS/2

On OS/2-based clients, invokable TPs are configured through entries in the SNA.INI file. The following table lists the section headings and environment variables used:

Type of TP	Section in SNA.INI listing TP names only	Section and possible environment variables defining TP
Autostarted invokable TP on a client running OS/2	**[SNAServerAutoTPs]** ***TPName1***=*SectionName1* ... ***TPNameX***=*SectionNameX*	**[*SectionName1*]** **PathName**=*path* **LocalLU**=*LUalias* **Parameters**=*ParameterList* **TimeOut**=*number* **Queued**={ YES | NO } **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }[1] ***Username1***=*Password1*[1] ... ***UsernameX***=*PasswordX*[1] **Environment**=*VariableList* **NewScreenGroup**={ 1 | 0 } **IconFile**=*path* **SessionType**=*number* **PgmControl**=*number* **InitXPos**=*number* **InitYPos**=*number* **InitXSize**=*number* **InitYSize**=*number*
Operator-started invokable TP on a client running OS/2	**[SNAServerAutoTPs]** ***TPNameN***=*SectionNameN* ... ***TPNameX***=*SectionNameX*	**[*SectionNameN*]** **LocalLU**=*LUalias* **TimeOut**=*number* **Queued**=OPERATOR **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }[1] ***Username1***=*Password1*[1] ... ***UsernameX***=*PasswordX*

Note

1 **AlreadyVerified** and Username/Password lines are used only if **ConversationSecurity** is set to YES.

This section contains:

- Environment Variables for OS/2-Based Clients
- Translating SNA Service TP Names to ASCII for SNA.INI

Environment Variables for OS/2-Based Clients

The following list shows the correct form for the sections and entries to add to the SNA.INI file (located in the root Host Integration Server 2000 directory) for autostarted invokable TPs on an OS/2-based client. The section headings are shown enclosed in square brackets; include the brackets when adding the section to the SNA.INI file.

For each TP type, the applicable variables and their locations are shown in Clients Running OS/2.

[SNAServerAutoTPs]

TPNameX=*SectionNameX*

For all TPs. Associates *TPnameX* with *SectionNameX*. Additional lines can follow *TPNameX*=*SectionNameX*, each one using the same syntax to name a different TP and the section containing the information for that TP.

[*SectionName*]

Forms a section heading for entries applying to one TP; *SectionName* must match a section name listed under **[SNAServerAutoTPs]**.

PathName=*path*

Specifies the full path and file name of the executable file. The default is TPNAME.EXE.

LocalLU=*LUalias*

Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters=*ParameterList*

Lists strings to be passed as command line parameters for the TP. Separate parameters with spaces. The default is no parameters.

Timeout=*number*

Specifies the time in milliseconds that a RECEIVE_ALLOCATE will wait before timing out. The default is infinity (no limit).

Queued={ YES | NO }

Queued=OPERATOR

Specifies the type of TP: YES for an autostarted queued TP, NO for an autostarted nonqueued TP, or OPERATOR for an operator-started TP (which must always be queued). The default is YES.

ConversationSecurity={ YES | NO }

Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified={ YES | NO }

Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For detailed information about the AlreadyVerified variable, see its description under Registry Entries for Clients Running Windows 2000 or Windows NT.

The default is NO.

Username1*=*Password1[1]

...

UsernameX*=*PasswordX[1]

Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can have as many as 10 characters each. Both parameters are case-sensitive. This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the **AlreadyVerified** entry.

Environment=*VariableList*

For an autostarted TP, lists the variables to be set in the TP's environment. Separate multiple variables with spaces.

NewScreenGroup={ 1 | 0 }

For an autostarted TP, specifies 1 to indicate that the TP runs in the foreground, or 0 to indicate that the TP runs in the background. The default is 1 (foreground).

IconFile=*path*

For an autostarted TP, specifies the full path and file name of the icon file. The default is no icon file.

SessionType=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 0.

PgmControl=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 32768.

InitXPos=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 80.

InitYPos=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 80.

InitXSize=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 470.

InitYSize=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 330.

SessionType, **PgmControl**, **InitXPos**, **InitYPos**, **InitXSize**, and **InitYSize** are filled directly into the **STARTDATA** structure passed to the OS/2 **DosStartSession** call for the new session.

Translating SNA Service TP Names to ASCII for SNA.INI

For SNA service TPs on Host Integration Server 2000 or SNA Server clients running OS/2, the line naming the TP in the SNA.INI file must specify the TP name in ASCII. The following paragraphs tell how to convert a TP name to this form. The line should be placed in the **[SNAServerAutoTPs]** section of the file, as shown in Clients Running OS/2.

An SNA service TP name is normally up to four bytes in length; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for SNA.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, a service TP name of 0x21 0xD7 0xD7 is equivalent to SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Clients Running MS-DOS

Host Integration Server 2000 and SNA Server do not support invokable TPs on MS-DOS–based clients.

Configuring TPs on Host Integration Server and SNA Server

The following topics describe how configuration of invoking and invokable TPs works on Microsoft Host Integration Server 2000 and Microsoft SNA Server 4.0.

This section contains:

Configuring Invoking TPs on Host Integration Server and SNA Server

For a server running Host Integration Server 2000 or SNA Server to support the beginning of the invoking process (that is, to accept the TP_STARTED and ALLOCATE or MC_ALLOCATE verbs issued by an invoking TP), the following parameters must be configured correctly:

1. If the invoking TP specifies in **TP_STARTED** the LU alias that it uses, that LU alias must match a local APPC LU alias on the supporting server running Host Integration Server or SNA Server. If the invoking TP leaves the LU alias blank in **TP_STARTED**, one of two methods for designating a default LU must be carried out on the supporting server running Host Integration Server or SNA Server:

   Assign a default local APPC LU to the user or group that starts the invoking TP (that is, the user or group logged on at the system from which **TP_STARTED** is issued).

   —or—

2. Designate one or more LUs as members of the default outgoing local APPC LU pool. The SNA server first attempts to determine the default local APPC LU of the user who started the TP, then attempts to assign an available LU from the default outgoing local APPC LU pool; if these attempts fail, the SNA server rejects the request.

   - In most situations, the supporting SNA server must contain an appropriate connection to another system (host or peer). Sometimes, for testing purposes, the server running Host Integration Server 2000 or SNA Server contains two local LUs paired together (for invoking and invokable TPs that are in the same domain); in this situation, a connection to a host or peer is not necessary.
   - If the invoking TP specifies in ALLOCATE or MC_ALLOCATE the partner LU alias, that LU alias must match an LU alias that is paired with the local LU alias specified in TP_STARTED. If the partner LU alias is left unspecified in ALLOCATE or MC_ALLOCATE, a default remote APPC LU must be assigned to the user who started the invoking TP. The default remote APPC LU is configuring using SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0. If the default remote APPC LU is used, it must be paired with the local LU that will be used. Otherwise, ALLOCATE or MC_ALLOCATE fails.

The preceding parameters support the beginning of the invoking process. For the invoking process to successfully complete, additional parameters must be configured as described in Configuring Invokable TPs on Host Integration Server and SNA Server.

Configuring Invokable TPs on Host Integration Server and SNA Server

For a computer running Host Integration Server 2000 or SNA Server to receive allocation requests from an invoking TP on another system and route those requests to an invokable TP, certain parameters must be configured correctly:

- The Host Integration Server or SNA Server must have a connection to the system from which the invoking TP's request is sent.
- The Host Integration Server or SNA Server must have a remote LU capable of receiving the incoming request. This remote LU can be configured either explicitly or implicitly.

  When configured explicitly, there is an explicit match between a remote LU alias on the Host Integration Server or SNA Server and the alias of the LU that conveys the invoking TP's request.

  When configured implicitly, an implicit incoming remote LU (with its implicit incoming mode) is used. This means that several items must work together. First, the LU alias specified in the incoming request (the LU alias requested for the invokable TP) must match a local LU alias on the Host Integration Server or SNA Server receiving the request. Second, the local LU on the Host Integration Server 2000 server must have an implicit incoming remote LU assigned to it. The properties of the implicit incoming remote LU will be used for that LU-LU session. For more details about how an implicit incoming remote LU works, see the *Microsoft Host Integration Server 2000 online books*.
- Appropriate local LUs must be defined in the Host Integration Server or SNA Server configuration. For descriptions of several ways to set up these local LUs, see Arranging TPs Within an SNA Network.

Arranging TPs Within an SNA Network

If your installation of Microsoft Host Integration Server or Microsoft SNA Server contains multiple systems (multiple clients and/or multiple SNA servers), you can place a given invokable TP on more than one system. When an invoking request is received in such an installation, there can be a choice of systems on which to run the invokable TP. You can maintain specific control over this choice; alternatively, by following the instructions in TP Name Not Unique; Local LU Alias Unspecified, you can allow Host Integration Server or SNA Server to make the choice randomly to distribute the load.

You can maintain specific control over the choice of system by setting up invokable TPs with unique names, or by setting up each invokable TP to run only with a specific, unique LU alias. With this arrangement, the information provided by the invoking TP (in the ALLOCATE or MC_ALLOCATE verb) specifies the system on which the invokable TP should run.

You can allow Host Integration Server or SNA Server to make the system choice randomly by setting the DloadMatchLocalFirst registry entry to NO and using invokable TPs that leave the local LU alias unspecified. Then, when an incoming request is received, it is routed randomly, rather than preferentially to the local Host Integration Server or SNA Server; in addition, no matter what LU alias is requested for the invokable TP, there cannot be a mismatch. Host Integration Server or SNA Server starts one instance of the requested TP, choosing randomly among the available systems.

The following topics describe some of the possible arrangements that can be made for running TPs.

This section contains:

TP Name Unique for Each TP

One way to specify the intended system where the invokable TP will run is to use a unique TP name for each invokable TP. In this arrangement, the invoking TP identifies the intended invokable TP (and system) simply by naming the TP. This makes it unnecessary for an invokable TP to specify any LU alias in registry or environment variables.

TP Name Not Unique; Local LU Alias Unique

Another way to specify the intended system where the invokable TP will run is to give the same name to multiple invokable TPs, but associate each TP with a unique local LU alias. To do this, configure each invokable TP (through registry or environment variables) to use a unique local LU alias. Then set up the invoking TPs so that each one is routed not only to the correct TP name but also to the correct partner LU alias for the intended invokable TP.

TP Name Not Unique; Local LU Alias Unspecified

If it does not matter on which system an invokable TP runs, use the same name for multiple invokable TPs and do not specify an LU alias in the registry or environment variables for the TPs. In this situation, there are no associated LU aliases in the list of available invokable TP names on a SNA server. Thus, a request received from an invoking TP cannot cause a mismatch on the LU alias, and will match according to the TP name.

In this situation, if you set the DloadMatchLocalFirst registry entry to NO, Host Integration Server OR SNA Server randomly routes the request to one of the available TPs. This spreads the processing load among multiple systems and provides hot backup (the ability to take systems online and offline without disrupting service).

Troubleshooting for Invokable TPs

If there are difficulties with starting an invokable TP, there may be a mismatch between the information for the invokable TP, the invoking TP, and/or LUs in the Host Integration Server or SNA Server configuration. That is, there may be a mismatch between the parameters for RECEIVE_ALLOCATE, TP_STARTED, ALLOCATE, or MC_ALLOCATE and/or LU aliases specified in server configuration. LU aliases are configured using SNA Manager on Host Integration Server 2000 and configured using SNA Explorer on SNA Server 4.0. For details about how to specify LU aliases see the *Microsoft Host Integration Server 2000 onine books*.

Simplifying APPC Configuration

There are several features in Host Integration Server 2000 and SNA Server that can simplify configuration for APPC:

- The implicit incoming remote LU and the implicit incoming mode, which allow Host Integration Server or SNA Server to accept requests that arrive by unrecognized remote LUs and modes.
- The default local APPC LU and the default remote APPC LU, which allow LU aliases to be associated with user or group names, simplifying the routing of incoming requests and the configuration of client systems.
- The default outgoing local APPC LU pool, which allows LUs to be allocated dynamically to any invoking TP that does not specify a local LU.
- Automatic partnering, which automatically creates LU-LU pairs and assigns modes to the pairs.

For more information about these features, see the *Microsoft Host Integration Server online books*.

Sync Point Level 2 Support in Host Integration Server and SNA Server

The following topics describe additions to Microsoft Host Integration Server 2000 and Microsoft SNA Server 4.0 that enable Sync Point Level 2 support to the LU 6.2 protocol stack.

This section assumes familiarity with the existing Host Integration Server or SNA Server APPC basic and mapped conversation interfaces. It does not attempt to explain the SNA formats and protocols for implementing Sync Point protocols, which are described in *SNA LU6.2 Reference: Peer Protocols* published by IBM (Document SC31-6808-1).

This section contains:

Sync Point Functional Overview

The Sync Point support additions to Host Integration Server 2000 and SNA Server 4.0 allow vendors to provide Sync Point services over LU 6.2 conversations provided by the LU 6.2 protocol stack in Host Integration Server or SNA Server 4.0. These additions do not implement the architected Sync Point components and TPs necessary for a complete Sync Point implementation. In particular, the following Sync Point components are not implemented, and must be provided by the vendor.

- Sync Point Services (SPS)
- Conversation-Protected Resource Manager (C-PRM)
- Resynchronization TP

These vendor-supplied components and applications are expected to implement the **SYNCPT** and **BACKOUT** verbs used for Sync Point services. The **SYNCPT** verb is used to synchronize transactions. The **BACKOUT** verb is used to back out of a transaction.

SPS, C-PRM, and the Resynchronization TP are specific components of the SNA Sync Point architecture described in *SNA LU6.2 Reference: Peer Protocols* published by IBM.

Host Integration Server 2000 and SNA Server 4.0 have been modified to add the features necessary to support these components, namely:

- Additions to the existing APPC API to support implementation of Sync Point verbs.
- Accounting support for Sync Point protocols.
- Modifications to invokable TP initiation.

Changes to the APPC basic and mapped conversation APPC APIs are made so as to ensure backward compatibility with existing APPC applications that adhere strictly to the API.

> Applications must zero all reserved verb control block (VCB) members before issuing an APPC verb. If this is not done, the application may inadvertently invoke one of the new APPC features.

Sync Point Support Architecture

The Sync Point support provided by Host Integration Server 2000 and SNA Server 4.0 assumes a particular implementation architecture by the vendor, as follows:

- The vendor provides a communication interface to its own clients requiring Sync Point Services (SPS).
- The vendor API maps its communication and Sync Point requests to the Host Integration Server or SNA Server APPC API.
- The vendor provides a single Microsoft® Windows 2000 or Microsoft® Windows NT® process, the Transaction Monitor, that is responsible for:
    - Issuing all APPC verbs.
    - Implementing the architected Resynchronization TP.
    - Implementing the architected Conversation-Protected Resource Manager (C-PRM) component of the LU.
    - Implementing the architected SPS component of the LU.

  The Transaction Monitor must reside on the same computer as the Host Integration Server or SNA Server containing the LUs for which it is providing Sync Point services. Both incoming and outgoing Sync Point conversations for this Transaction Monitor will be routed through this Host Integration Server 2000 server only.

Detailed descriptions of the three architected Sync Point components can be found in *SNA LU6.2 Reference: Peer Protocols* published by IBM.

Sync Point Session Support

This section discusses support for Sync Point session activation and deactivation in Host Integration Server 2000 and SNA Server 4.0.

This section contains:

Sync Point Session Activation

If Host Integration Server 2000 or SNA Server 4.0 are to support Sync Point conversations, this must be specified at session activation time. The configuration of Host Integration Server 2000 and SNA Server 4.0 is modified to allow the system administrator to specify which (if any) local LUs will be used for Sync Point conversations.

The Local LU Configuration property page on Host Integration Server or SNA Server contains a new check box. When checked, it indicates that the local LU can participate in Sync Point sessions. Host Integration Server or SNA Server uses this option to determine the parameters it sends on BIND requests and responses.

When Host Integration Server or SNA Server initiates an LU 6.2 session on an LU designated as supporting Sync Point, it sets the synchronization level on BIND requests to indicate that the session can support Sync Point and Backout. If the partner LU also supports Sync Point and Backout, the session is available for conversations requiring Sync Point support. If the partner LU does not support Sync Point, the session will not be used for Sync Point conversations.

Similarly, if the local LU is configured for Sync Point and the partner LU's BIND request indicates that it supports Sync Point, Host Integration Server or SNA Server sends BIND responses specifying that Sync Point is supported. In this case, the session can be used for Sync Point conversations.

Sync Point Session Deactivation

A Sync Point implementation needs to determine whether it has lost connectivity to a partner when establishing Sync Point conversations so that it can know whether or not to resynchronize. To obtain this information, Host Integration Server and SNA Server provide a new APPC verb, GET_LU_STATUS that reports the status of a particular remote LU. The information returned by this verb is as follows:

- Current number of active LU 6.2 sessions between the remote LU and the TP's local LU.
- Whether or not the number of active sessions dropped to zero at any time since this verb was last issued for the remote LU.

Note that the zero sessions indicator is reset to AP_NO each time the verb is issued by any process. It is therefore imperative that only one process issues this verb; otherwise information may be lost.

Starting Local Sync Point TPs

Local TPs are created by issuing the TP_STARTED verb to Host Integration Server or SNA Server. The **TP_STARTED** verb has been modified by adding the new VCB member **syncpoint_rqd** to allow a TP to specify that it requires Sync Point services.

By setting syncpoint_rqd to AP_YES, a TP indicates that it requires Sync Point services from Host Integration Server or SNA Server 4.0. A value of AP_NO (the default) indicates that Sync Point services are not required.

Since this member cannot be incorporated within the existing TP_STARTED VCB, the TP must use a larger VCB structure. To indicate that the VCB is longer than usual, the opext member of the VCB must be combined using OR with the value AP_EXTD_VCB before calling APPC.

If the TP is started on a server running SNA Server earlier than version 3.0 with the opext member of the VCB set for Sync Point support, the TP_STARTED verb will fail with a primary return code of AP_INVALID_OPCODE.

Conversations started by TPs requiring Sync Point support will be routed only by the Host Integration Server or SNA Server software running on the same computer. They will not be routed to other LAN-attached servers.

Sync Point Conversation Activation

This section discusses support for Sync Point conversation activation in Host Integration Server and SNA Server 4.0.

This section contains:

Locally Initiated Conversations

Conversations are initiated locally by issuing an ALLOCATE or MC_ALLOCATE verb. The **ALLOCATE** and **MC_ALLOCATE** verbs are modified to support additional parameters required by Sync Point support. The supplied **synclevel** parameter of the **ALLOCATE** and **MC_ALLOCATE** verbs can take on a value of AP_SYNCPT, which specifies that the conversation requested is a Sync Point conversation.

Remotely Initiated Conversations

Applications wishing to receive remotely initiated conversations (incoming Attaches) issue a RECEIVE_ALLOCATE verb. To accommodate Sync Point support, the **RECEIVE_ALLOCATE** verb is modified in a number of ways as follows:

- The returned **sync_level** parameter of the **RECEIVE_ALLOCATE** verb can take on a value of AP_SYNCPT, specifying that the conversation is a Sync Point conversation. The value of the **sync_level** parameter can also be determined by issuing a GET_ATTRIBUTES verb on the new conversation.
- Support is added for the receipt of program initiation parameters (PIP) data by a new parameter to the RECEIVE_ALLOCATE verb:

  The **pip_incoming** parameter is set by the application to indicate whether it is willing to accept incoming PIP data, and is returned by Host Integration Server or SNA Server to indicate whether PIP data is available to be received. If the application does not wish to receive PIP data, this member should be set to AP_NO, the default, before issuing the **RECEIVE_ALLOCATE** verb. If it is willing to accept PIP data, this member should be set to AP_YES. On completion of the **RECEIVE_ALLOCATE** verb, this member will be set to AP_YES if PIP data is available to be received by the application and to AP_NO otherwise.

- If PIP data is available, the application can receive it by issuing one of the verbs for receiving data on completion of the RECEIVE_ALLOCATE verb. For basic conversations, these receive verbs include RECEIVE_AND_POST, RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE. On basic conversations the PIP data will be returned inclusive of the general data stream (GDS) header for PIP data (GDS identifier 0x12F5). For mapped conversations, these receive verbs include MC_RECEIVE_AND_POST, MC_RECEIVE_AND_WAIT, and MC_RECEIVE_IMMEDIATE. On mapped conversations, Host Integration Server 2000 removes the 4-byte GDS header, and returns the PIP data only.
- For basic conversations, if the application issues a SEND_ERROR, DEALLOCATE, or TP_ENDED verb before the PIP data is received, the PIP data will be discarded. For mapped conversations, if the application issues an MC_SEND_ERROR, MC_DEALLOCATE, or TP_ENDED verb before the PIP data is received, the PIP data will be discarded.
- If PIP data is received for a TP that cannot or does not want to receive it, the conversation is rejected with a primary return code of AP_ALLOCATION_ERROR, and a secondary return code of AP_PIP_NOT_ALLOWED.

Already Verified Support

In an implementation where a Host Integration Server or SNA Server application acts as a gateway between an SNA network and a non-SNA network, it is possible that non-Host Integration Server or SNA Server clients of the gateway may require Sync Point Level 2 conversation security. Since the originating client will have validated the relevant user identifier and password, the gateway application should specify conversation security of AP_SAME when starting a conversation on behalf of the client. In this case, however, Host Integration Server and SNA Server assume that the user identifier to be used has previously been received on an Attach targeted at the TP. In the case of a non-Host Integration Server or SNA Server client this is not the case.

To allow such a gateway to support Sync Point Level 2 conversation security, Host Integration Server and SNA Server provide a new verb, SET_TP_PROPERTIES that allows the gateway application to set the user identifier for the TP before allocating a conversation with security of AP_SAME. This verb will normally be issued once, immediately after TP_STARTED, to set the user identifier for all the TP's conversations.

Presentation Header Support in Data Transfers

For basic conversations, Sync Point commands are sent by means of presentation headers (PS) across LU 6.2 conversations using the SEND_DATA or MC_SEND_DATA verb. All presentation headers contain length fields that specify a length of 1, which is usually illegal. To support Sync Point conversations, the following modifications are made to the Host Integration Server or SNA Server presentation services component:

- On basic conversations with a **synclevel** of AP_SYNCPT, data transferred specifying a general data stream (GDS) variable length of 1 will not be rejected. If the **synclevel** is not AP_SYNCPT, they will be rejected as before.
- On mapped conversations, PS headers will not be wrapped as mapped conversation application data logical records (with GDS identifier 0x12FF) when they are sent, or have the GDS header stripped off when they are received.
- On mapped conversations, it is the responsibility of the application to provide the compete PS header including the length field. Similarly, the length field will be included in PS header data returned by receive verbs.

To achieve the latter the MC_SEND_DATA verb and the receive verbs (MC_RECEIVE_AND_POST, MC_RECEIVE_AND_WAIT, and MC_RECEIVE_IMMEDIATE) require modifications as follows:

- A new parameter, **data_type**, is added to the **MC_SEND_DATA** verb. When this is set to AP_APPLICATION (the default, 0x00), the data is sent as application data (GDS identifier 0x12FF) as usual. When it is set to AP_PS_HEADER, the data is sent as described above.
- The following two new values are added for the what_rcvd member of the receive verbs to specify that the received data is a PS header:

  AP_PS_HEADER_COMPLETE

  AP_PS_HEADER_INCOMPLETE

- If an application issues a receive verb with **rtn_status** set to AP_YES, Host Integration Server or SNA Server will return status in combination with AP_PS_HEADER_COMPLETE, with the exception of AP_DEALLOCATE_NORMAL and AP_CONFIRM_DEALLOCATE. This is to prevent the conversation being prematurely disconnected from the LU 6.2 session when a COMMIT PS header arrives with the end of conversation indication.

It is the responsibility of the vendor-supplied Sync Point support component to convert these PS headers into the appropriate Sync Point return codes (for example, TAKE_SYNCPT).

User Control Data

For mapped conversations, the MC_SEND_DATA verb and the receive verbs (MC_RECEIVE_AND_POST, MC_RECEIVE_AND_WAIT, and MC_RECEIVE_IMMEDIATE) are modified to allow applications to send and receive data in user control data general data stream (GDS) variables instead of the regular application data GDS variables. The **MC_SEND_DATA** verb is modified as follows:

- A new parameter, **data_type**, is added. When **data_type** is set to AP_USER_CONTROL_DATA, the data is sent as user control data (GDS identifier 0x12F2). When it is set to AP_APPLICATION (the default), the data is sent as application data (GDS identifier 0x12FF). Note that the APPC library automatically creates the GDS header on behalf of the application for both AP_APPLICATION and AP_USER_CONTROL_DATA data records.
- The mapped conversation receive verbs are modified to allow applications to receive user control data by adding two new values for the what_rcvd parameter, as follows:

  AP_USER_CONTROL_DATA_COMPLETE

  AP_USER_CONTROL_DATA_INCOMPLETE

Implied Forget

LU 6.2 Sync Point sessions can use an optimization of the architected message flows known as implied forget. When the protocol specifies that a FORGET presentation header (PS) is required, the next data flow on the session implies that a FORGET has been received, even though it has not. In the normal situation, the TP is aware of the next data flow when data is received or sent on one of its Sync Point conversations.

However it is possible that the last message that flows is caused by the conversation being deallocated. In this case, the TP is unaware when the next data flow on the session occurs. To provide the TP with this notification, the DEALLOCATE and MC_DEALLOCATE verbs are modified to allow the TP to register a callback function which will be called:

- On the first normal flow transmission (request or response) over the session used by the conversation.
- If the session is unbound before any other data flows.
- If the session is terminated abnormally due to a data link control (DLC) outage.

The callback procedure can take any name because the address of the procedure is passed into the APPC DLL.

Note that the DEALLOCATE and MC_DEALLOCATE verbs will probably complete before the callback routine is called. The conversation is considered to be in RESET state and no further verbs can be issued using the conversation identifier. If the application issues a TP_ENDED verb before the next data flow on the session, the callback routine will not be invoked.

The DEALLOCATE and MC_DEALLOCATE verbs are modified as follows to support implied forget:

- A new member, **callback**, is added to allow the TP to specify the address of the function to call on the next data flow on the session being used by the conversation being deallocated. If this member is NULL, no notification will be provided. A vendor would normally supply this callback function.
- The DEALLOCATE and MC_DEALLOCATE verbs also contain a correlator member which is returned as one of the parameters when the callback function is invoked. The application can use this parameter in any way (for example, as a pointer to a control block within the application).

Host Integration Server and SNA Server allow TPs to deallocate conversations immediately after sending data by specifying the **type** member in the SEND_DATA and MC_SEND_DATA verbs as AP_SEND_DATA_DEALLOC_FLUSH, AP_SEND_DATA_DEALLOC_SYNC_LEVEL, AP_SEND_DATA_DEALLOC_ABEND, and AP_SEND_DATA_DEALLOC_CONFIRM. However, the **SEND_DATA** and **MC_SEND_DATA** verbs do not contain the implied forget callback function. TPs wishing to receive implied forget notification must issue a DEALLOCATE or MC_DEALLOCATE verb explicitly.

Sync Point Level 2 Confirm Support

The current APPC implementation in Host Integration Server and SNA Server supports conversations with **synclevel** of AP_NONE, AP_CONFIRM_SYNC_LEVEL, or AP_SYNCPT. The DEALLOCATE, MC_DEALLOCATE, PREPARE_TO_RECEIVE, and MC_PREPARE_TO_RECEIVE verbs specify a **type** member indicating the synchronization level required. This parameter is interpreted as follows:

Allocated synclevel	Type specified	Action performed
AP_NONE	AP_FLUSH	Action of FLUSH or MC_FLUSH verb before deallocation or change of direction.
AP_NONE	AP_SYNCLEVEL	Action of **FLUSH** or **MC_FLUSH** verb before deallocation or change of direction.
AP_SYNCPT	AP_FLUSH	Action of FLUSH or MC_FLUSH verb before deallocation or change of direction.
AP_SYNCPT or AP_CONFIRM_SYNC_LEVEL	AP_CONFIRM_TYPE	Action of CONFIRM or MC_CONFIRM verb before deallocation or change of direction.
AP_SYNCPT	AP_SYNCLEVEL	It is assumed that a Sync Point implementation built using the APPC API in Host Integration Server or SNA Server implements the defer states appropriately. See the note below.

With an allocated **synclevel** of AP_SYNCPT and a specified **type** of AP_SYNCLEVEL, it is assumed that a vendor-supplied Sync Point component implements the defer states appropriately. A vendor-supplied Sync Point system must:

- Intercept DEALLOCATE, MC_DEALLOCATE, PREPARE_TO_RECEIVE, and MC_PREPARE_TO_RECEIVE verbs on Sync Point Level 2 conversations when type AP_SYNCLEVEL is specified for **synclevel**.
- Maintain the defer state until one of the verbs valid in that state completes.
- On completion of the verb, issue the original DEALLOCATE, MC_DEALLOCATE, PREPARE_TO_RECEIVE, or MC_PREPARE_TO_RECEIVE verb to Host Integration Server 2000.

Host Integration Server and SNA Server do not implement the defer states directly. In particular, when a **DEALLOCATE**, **MC_DEALLOCATE**, **PREPARE_TO_RECEIVE**, or **MC_PREPARE_TO_RECEIVE** verb is received with a **type** specified as AP_SYNCLEVEL on a Sync Point conversation, this is treats as if the conversation has a **synclevel** of AP_NONE.

So that Sync Point Level 2 conversations can use confirm type synchronization, the DEALLOCATE, MC_DEALLOCATE, PREPARE_TO_RECEIVE, and MC_PREPARE_TO_RECEIVE verbs are modified to support a type member of AP_CONFIRM_TYPE.

Versions of SNA Server prior to SNA Server 3.0 support conversations with synclevel of AP_NONE or AP_CONFIRM_SYNC_LEVEL. The DEALLOCATE, MC_DEALLOCATE, PREPARE_TO_RECEIVE, and MC_PREPARE_TO_RECEIVE verbs specify a type member indicating the synchronization level required. This parameter is interpreted as follows:

Allocated synclevel	Type specified	Action performed
AP_NONE	AP_FLUSH	Action of FLUSH or MC_FLUSH verb before deallocation or change of direction.
AP_NONE	AP_SYNCLEVEL	Action of **FLUSH** or **MC_FLUSH** verb before deallocation or change of direction.
AP_CONFIRM_SYNC_LEVEL	AP_FLUSH	Action of **FLUSH** or **MC_FLUSH** verb before deallocation or change of direction.
AP_CONFIRM_SYNC_LEVEL	AP_SYNCLEVEL	Action of CONFIRM or MC_CONFIRM verb before deallocation or change of direction.

Sync Point Backout Support

This section describes backout support for Sync Point conversations.

This section contains:

Additional Sync Point Return Codes

When a remote TP issues a **BACKOUT** verb, the backout is reported to the local TP as a new primary return code value, AP_BACKED_OUT, on the next (current) verb issued. The local TP is provided access to the sense code information contained in the Backout FMH-7 by setting the **secondary_rc** field as follows:

- AP_BO_NO_RESYNC for sense code 0x08240000
- AP_BO_RESYNC for sense code 0x08240001

This new return code will only be supplied on conversations with **synclevel** AP_SYNCPT, and therefore will not be presented to existing applications.

The verbs on which this new return code can be returned are:

CONFIRM

MC_CONFIRM

MC_PREPARE_TO_RECEIVE

MC_RECEIVE_AND_POST

MC_RECEIVE_AND_WAIT

MC_RECEIVE_IMMEDIATE

MC_SEND_DATA

MC_SEND_ERROR

PREPARE_TO_RECEIVE

RECEIVE_AND_POST

RECEIVE_AND_WAIT

RECEIVE_IMMEDIATE

SEND_DATA

SEND_ERROR

Sending Backout on Sync Point Conversations

To send a Backout, an FMH-7 containing a sense code of 0x08240000 or 0x08240001 is sent on the session. This is done using the SEND_ERROR or MC_SEND_ERROR verb. To allow Host Integration Server 2000 to send the appropriate sense data, the **SEND_ERROR** and **MC_SEND_ERROR** verbs are modified as follows:

- A new field, **err_type**, is added to allow the TP to specify the type of error. The default is AP_PROG (0x00), which means existing TPs will continue to work unmodified.
- The err_type field in both verbs can take one of two new values, specifying the sense codes to be generated by Host Integration Server 2000:

  AP_BACKOUT_NO_RESYNC for sense code 0x08240000

  AP_BACKOUT_RESYNC for sense code 0x08240001

LUWID, Conversation Correlators, and Session Identifiers

The logical unit-of-work identifier (LUWID), conversation correlators, and session identifiers are important for all Sync Point operations and accounting purposes. The following sections describe how Host Integration Server and SNA Server 4.0 provide access to these components and, where appropriate, facilities to modify this information.

This section contains:

Generating and Setting LUWIDs

The LUWID is used to identify conversations that are part of a single Sync Point transaction. All conversations with the same LUWID are committed (or backed out) at the same time.

Host Integration Server and SNA Server assign two LUWIDs to a transaction program when the TP is started. For locally started TP's, this is when the TP_STARTED verb is issued. (In previous versions of SNA Server 2.*x* only one LUWID was assigned.) The first LUWID is the TP's protected LUWID. It is used by Host Integration Server 2000 as the LUWID for all synclevel AP_SYNCPT conversations allocated by the TP. When the TP issues an ALLOCATE or MC_ALLOCATE verb with a synclevel of AP_SYNCPT, Host Integration Server and SNA Server generate an Attach containing the TP's current protected LUWID.

The second LUWID is the TP's unprotected LUWID. It is used on all conversations allocated by the TP with a synclevel other than AP_SYNCPT.

For remotely initiated TPs, the incoming Attach may contain an LUWID for the TP—it is mandatory if the conversation has a synclevel of AP_SYNCPT. For Sync Point conversations, Host Integration Server and SNA Server save the LUWID as the TP's protected LUWID and generate a new unprotected LUWID for it. For conversations with a synclevel other than Sync Point (AP_SYNCPT), Host Integration Server and SNA Server save the LUWID as the TP's unprotected LUWID and generate a new protected LUWID.

Host Integration Server and SNA Server generate LUWIDs by concatenating the following:

- The fully qualified name of the local LU, preceded by a single byte indicating its length (exclusive of the length byte).
- A 6-byte LUW instance number, generated from the current date and time (modified to ensure uniqueness if necessary).
- A 2-byte LUW sequence number, initialized to 1.

If the fully qualified LU name component of the LUWID is not 17 bytes long, Host Integration Server and SNA Server do not add any padding between it and the LUW instance number. The application can determine the length of the LUWID, and the offsets within it of the LUW instance number and LUW sequence number, by examining the first byte of the LUWID, which indicates the length of the fully qualified LU name.

When Host Integration Server and SNA Server generate both a protected and an unprotected LUWID for a TP, the unprotected LUWID is created by incrementing the protected LUWID's instance number.

The protected LUWID needs to be changed by a TP for one of four reasons:

- When a transaction is backed out or committed, the LUWID sequence number must be incremented.
- If the transaction tree is split, a new LUWID must be generated for the TP.
- If the application uses multiple logical TPs to implement a transaction, each TP must have the same LUWID (different from that assigned by Host Integration Server or SNA Server).
- If the application is acting as a gateway from a non-SNA environment and LUWIDs are received by a means other than an Attach.

To allow a TP to set or generate new LUWIDs, a new verb, SET_TP_PROPERTIES, is provided by the APPC API. This verb allows the TP to either set its LUWIDs to an existing value, by providing the LUWIDs, or generate new ones and use them from then on. When a new LUWID is generated by Host Integration Server or SNA Server, it is guaranteed to be unique.

Note that it is the responsibility of the application (the Sync Point system component) to transmit the new LUWID PS header to the partner Sync Point system when the protected LUWID is changed. Similarly, when a new LUWID PS header is received, the application must inform the LU by issuing SET_TP_PROPERTIES.

Extracting LUWIDs

Both LUWIDs for a particular TP can be determined by issuing the GET_TP_PROPERTIES verb. The **GET_TP_PROPERTIES** verb returns the TP's unprotected LUWID in the **luw_id** field.

If the TP needs to access the protected LUWID, it must combine the opext member of the VCB with the value AP_EXTD_VCB using OR before issuing the verb. The protected LUWID will then be returned in the prot_luw_id field. If the opext field does not contain the AP_EXTD_VCB bit, the verb control block is presumed to end immediately before the prot_luw_id field.

The LUWID for a particular conversation can be determined by issuing a GET_ATTRIBUTES or MC_GET_ATTRIBUTES verb on the conversation. These verbs are modified as follows:

- A new field, **luw_id**, is added in which the LUWID is returned. The LUWID returned is the protected one if the conversation was allocated with **synclevel** field of the ALLOCATE or MC_ALLOCATE verb set to Sync Point (AP_SYNCPT); otherwise it is the unprotected one.
- Since the luw_id field cannot be incorporated within the existing verb control blocks, the TP must use a larger VCB structure. To indicate that the VCB is longer than usual, the opext field of the VCB must be combined with the value AP_EXTD_VCB using OR before calling APPC.
- The sync_level field of the GET_ATTRIBUTES or MC_GET_ATTRIBUTES verb can take an additional value, AP_SYNCPT, when the conversation was allocated with the synclevel field of the ALLOCATE or MC_ALLOCATE verb of Sync Point (AP_SYNCPT).

Session Identifiers

Host Integration Server and SNA Server maintain a unique identification for every LU 6.2 session it has with a remote LU. This 8-byte identifier is generated by Host Integration Server or SNA Server every time it starts a new session (or is received by Host Integration Server or SNA Server when a session is initiated remotely). The Sync Point resynchronization protocols require knowledge of the session identifier.

To provide this, the MC_GET_ATTRIBUTES and the GET_ATTRIBUTES verbs have been modified to return the session identifier of the session over which a particular conversation is allocated. The MC_GET_ATTRIBUTES and GET_ATTRIBUTES verbs can be used to retrieve this sess_id field of the VCB if the opext field of the VCB is combined with the value AP_EXTD_VCB using OR before calling APPC.

Configuration Changes for Sync Point Support

A new check box is added to the Local LU Configuration dialog box. When checked, this indicates that the local LU is able to participate in synclevel Sync Point sessions. Host Integration Server and SNA Server use this option to determine the synclevel BIND parameters it sends on BIND requests and responses.

This field is added to the Host Integration Server and SNA Server configuration file in a field that is no longer used by Host Integration Server or SNA Server. Existing configurations from earlier versions of SNA Server will therefore continue to work unmodified.

Accepting Incoming Attaches

The Sync Point support in Host Integration Server and SNA Server is intended for use only by gateway applications that implement the architected SNA Sync Point components, including Conversation-Protected Resource Manager (C-PRM). In a Sync Point implementation, it is necessary for C-PRM to be aware of all protected conversations, both locally initiated and remotely initiated. This can be achieved in Host Integration Server or SNA Server by C-PRM intercepting the conversation allocation and deallocation verbs and issuing them on behalf of the transaction program (TP). Note that since Host Integration Server and SNA Server do not allow TP or conversation identifiers to be shared across processes, this also means that the process containing C-PRM must also intercept all APPC verbs issued by the client TPs.

For locally initiated TPs, this is straightforward. However for incoming Attaches, the situation is made more complex by the requirement that the RECEIVE_ALLOCATE verb specify the name of the TP to be matched with the Attach.

In some implementations, this will not be an issue, as the gateway will be aware of the names of all the transactions passing through it. To support this situation, the RECEIVE_ALLOCATE verb has been enhanced as described in the following topic to permit the gateway to indicate that it can accept Sync Point conversations.

In other implementations, the gateway does not know the names of the transactions passing through it. This is particularly so when the gateway is providing a conversion between SNA and another communications protocol. In this case, Host Integration Server and SNA Server allow the gateway process to register itself as a Sync Point Attach Service, indicating that it is willing to accept incoming Attaches for any Sync Point conversation. In this case, the gateway must be implemented as a Sync Point Attach Manager.

This section contains:

- Sync Point Knows Transaction Names
- Sync Point Attach Manager
- Rejecting Remotely Initiated Conversations

Sync Point Knows Transaction Names

A Sync Point implementation that knows the names of all the transactions that can be supported (for example, through configuration of the gateway) may accept incoming Sync Point conversations by issuing a RECEIVE_ALLOCATE verb specifying the name of the transaction and indicating that it is willing to accept Sync Point conversations.

The RECEIVE_ALLOCATE verb was modified to allow a TP to specify that it can accept Sync Point conversations by adding a new syncpoint_rqd field to the VCB. When this field is set to AP_YES it indicates that the transaction program can accept Sync Point conversations from Host Integration Server or SNA Server. When this field is set to AP_NO (the default), it indicates that Sync Point conversations are not supported.

Sync Point Attach Manager

Instead of issuing separate RECEIVE_ALLOCATE verbs for each possible transaction name, a Sync Point implementation may instead register as the Sync Point Attach Manager for Host Integration Server or SNA Server. It does so by issuing a **RECEIVE_ALLOCATE** verb specifying a TP name consisting of all 0x00s.

When a Sync Point Attach Manager is registered, the following changes are effected in server's incoming Attach support on Host Integration Server or SNA Server:

- When an Attach message arrives for any TP name on a conversation with the **syncpoint_rqd** field of the VCB set to AP_YES, Host Integration Server 2000 matches it with the application that issued the special **RECEIVE_ALLOCATE** verb registering itself as the Sync Point Attach Manager.
- Any Attach message arriving for the Resynchronization TP (0x06F2) will automatically be routed to the Sync Point Attach Manager.
- If no RECEIVE_ALLOCATE has been issued for the Sync Point Attach Manager, or for the specific TP name, Host Integration Server or SNA Server will queue the Attach for a configured period of time. If no RECEIVE_ALLOCATE is issued in that time, the Attach will be rejected with a return code of TP_NOT_AVAILABLE_RETRY.
- If a RECEIVE_ALLOCATE is matched with the Attach message, the verb is returned to the TP with the tp_name field of the VCB set to the TP name contained in the Attach message.

Applications using this feature must adhere to two restrictions:

- All verbs issued on conversations started in this manner must be issued by the same process, as Host Integration Server or SNA Server cannot pass **tp_id**s between processes.
- Only a single process may register as the Sync Point Attach Manager on any server running Host Integration Server 2000 or SNA Server. If a second process attempts to register, its RECEIVE_ALLOCATE verb will return immediately with the primary return code set to AP_SYNCPOINT_MANAGER_ACTIVE.

Sync Point Attach Manager applications must reside on a Host Integration Server 2000 server. They may not be distributed across Host Integration Server or SNA Server clients. This restriction is imposed to ensure that only a single instance of Sync Point Services (SPS) and Conversation-Protected Resource Manager (C-PRM) exists for each LU on the Host Integration Server or SNA Server (which might not be the case if Sync Point Attach Managers were visible from multiple servers in the Host Integration Server or SNA Server domain).

The structure of the RECEIVE_ALLOCATE verb control block does not require modification to support this function.

Rejecting Remotely Initiated Conversations

In an environment where a Sync Point Attach Manager is receiving all Attach messages as described above, it may be necessary for it to reject an Attach for a particular TP name, either because the TP name is not valid or because there is another problem with the received Attach message. To enable the application to generate the correct return code at the initiating TP, the DEALLOCATE and MC_DEALLOCATE verbs are enhanced with new **deallocate_type** field values in the VCB that allow the application to specify the return code to be sent to the initiating TP. The new values for **deallocate_type** are:

AP_TP_NOT_AVAIL_RETRY

AP_TP_NOT_AVAIL_NO_RETRY

AP_TPN_NOT_RECOGNIZED

AP_PIP_DATA_NOT_ALLOWED

AP_PIP_DATA_INCORRECT

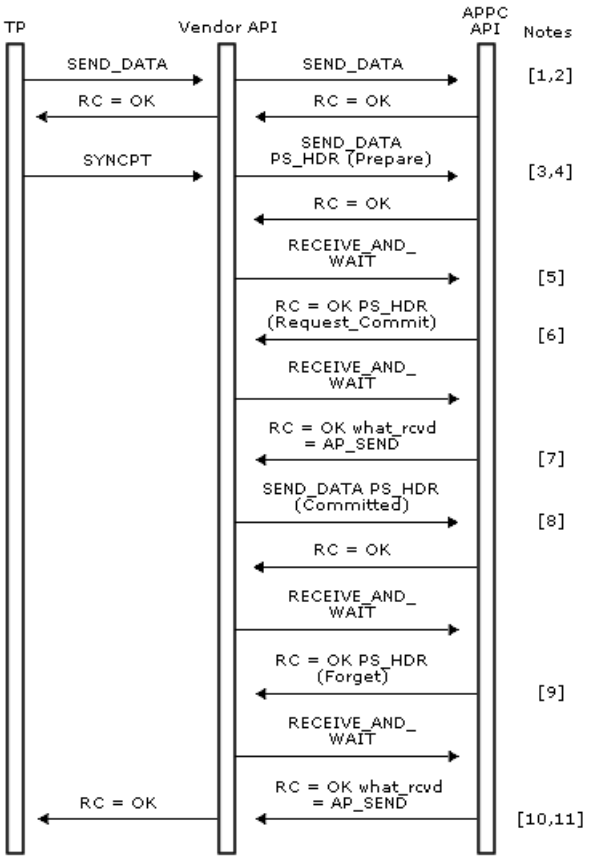
Sync Point Examples

This section contains example verb sequences for implementing the architected Sync Point verbs using the Sync Point facilities provided by Host Integration Server and SNA Server 4.0.

In the following figures, TP is the transaction program that requires Sync Point services. Vendor API is the vendor-supplied APPC API. This component provides the SPS and C-PRM components and a mapping between the vendor's APPC syntax and that of Host Integration Server or SNA Server. APPC API is the Host Integration Server or SNA Server APPC basic and mapped conversation interface.

This section contains:

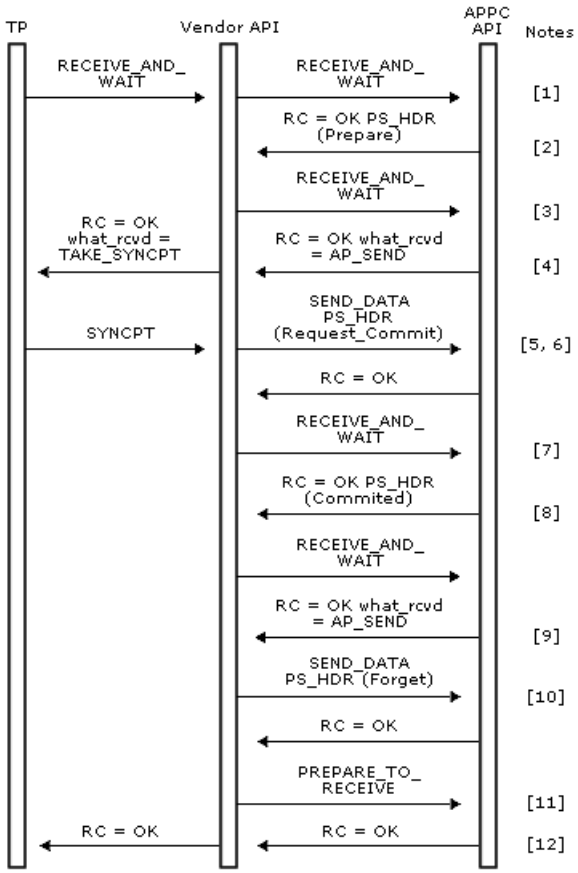
SYNCPT Verb Issued Locally



Notes

1. The transaction program issues a SEND_DATA or MC_SEND_DATA verb depending on whether a basic or mapped conversation is being used.
2. The SEND_DATA or MC_SEND_DATA VCB is passed transparently through the vendor API to Host Integration Server 2000. When the verb completes, the return code from Host Integration Server 2000 is returned to the transaction program.
3. The transaction program issues a SYNCPT verb to the vendor API.
4. The vendor API creates a PREPARE PS header and transmits it by issuing a SEND_DATA or MC_SEND_DATA verb. For a mapped conversation, the data_type field of the MC_SEND_DATA VCB must be set to AP_PS_HEADER.
5. On completion of the SEND_DATA or MC_SEND_DATA verb, the vendor API issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb.
6. The RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb completes with the what_rcvd field of the VCB with a value of AP_PS_HEADER. The data buffer is filled with the received REQUEST_COMMIT PS header.
7. Another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb is issued to get send direction. Note that the vendor API can combine these two verbs into a single request by setting the rtn_status field of the VCB to AP_YES in order to receive status with data on the first RECEIVE_AND_WAIT. or MC_RECEIVE_AND_WAIT.
8. A COMMITTED PS header is then transmitted using a SEND_DATA or MC_SEND_DATA verb.
9. The Vendor API issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb to receive the FORGET PS header from the remote TP.
10. Another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb is issued with the what_rcvd field of the VCB set to AP_SEND to get send direction (again the rtn_status RECEIVE_AND_WAIT field of the VCB can be set to AP_YES to combine these two verbs).
11. When send indication is received, the vendor API returns the SYNCPT verb to the local transaction program with an OK return code.

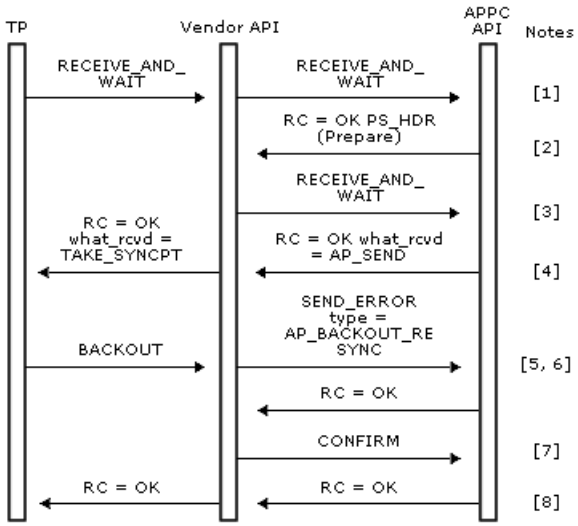
SYNCPT Verb Issued Remotely



Notes

1. The local TP issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb (depending on whether a basic or mapped conversation is being used) to receive data from the remote transaction program. The vendor API passes the verb transparently to Host Integration Server or SNA Server.
2. The **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** verb completes with **what_rcvd** = AP_PS_HEADER. The data buffer contains a PREPARE PS header.
3. Another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb is issued by the vendor API to receive the send indication from the remote TP.
4. The vendor API returns the transaction program's RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb with the what_rcvd field of the VCB set to TAKE_SYNCPT.
5. The transaction program issues a SYNCPT verb.
6. The vendor API generates a REQUEST_COMMIT PS header and transmits it using a SEND_DATA or MC_SEND_DATA verb. If the conversation is mapped, the MC_SEND_DATA verb is issued with the data_type field of the VCB set to AP_PS_HEADER.
7. The vendor API then issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb to give the remote TP direction to send.
8. The RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb completes with the what_rcvd field of the VCB set to AP_PS_HEADER. The data buffer contains a COMMITTED PS header.
9. Another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb is issued to get permission to send.
10. A FORGET PS header is prepared and sent to the remote transaction program.
11. The FORGET is flushed and direction given to the remote transaction program by issuing a PREPARE_TO_RECEIVE or MC_PREPARE_TO_RECEIVE with the ptr_type field of the VCB set to AP_FLUSH.
12. When the PREPARE_TO_RECEIVE or MC_PREPARE_TO_RECEIVE verb completes, the vendor API returns the SYNCPT verb to the local transaction program.

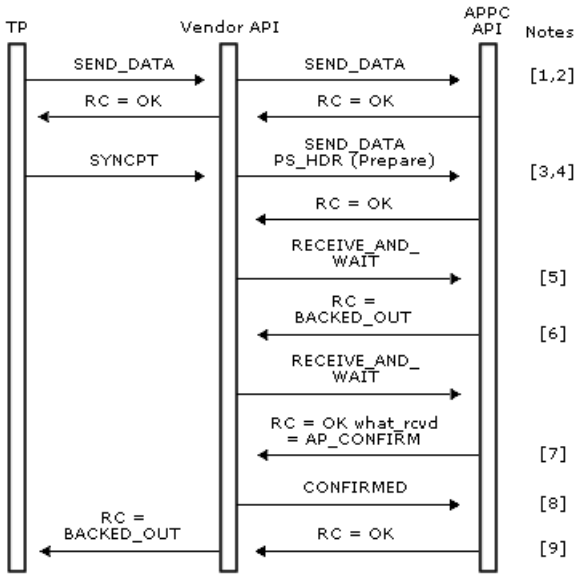
BACKOUT Verb Issued Locally



Notes

1. The local transaction program issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb (depending on whether a basic or mapped conversation is being used) to receive data from the remote transaction program. The vendor API passes the verb transparently to Host Integration Server or SNA Server.
2. The **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** verb completes with the **what_rcvd** field of the VCB set to AP_PS_HEADER. The data buffer contains a PREPARE PS header.
3. Another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb is issued by the vendor API to receive the send indication from the remote TP.
4. The vendor API returns the transaction program's RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb with the what_rcvd field of the VCB set to TAKE_SYNCPT.
5. The transaction program issues a BACKOUT verb to back out the transaction.
6. The vendor API generates a SEND_ERROR or MC_SEND_ERROR verb of type BACKOUT_RESYNC to send the Backout sense code 0x08240001.
7. The vendor API then issues a CONFIRM or MC_CONFIRM verb to flush the SEND_ERROR or MC_SEND_ERROR verb and request a response from the remote transaction program.
8. The CONFIRM or MC_CONFIRM verb completes when the remote transaction program issues a CONFIRMED or MC_CONFIRMED verb. The vendor API then returns the BACKOUT verb to the local transaction program.

BACKOUT Verb Issued Remotely



Notes

1. The transaction program issues a SEND_DATA or MC_SEND_DATA verb depending on whether a basic or mapped conversation is being used.
2. The **SEND_DATA** or **MC_SEND_DATA** VCB is passed transparently through the vendor API to Host Integration Server or SNA Server. When the verb completes the return code from Host Integration Server or SNA Server is returned to the transaction program.
3. The transaction program issues a SYNCPT verb to the vendor API.
4. The vendor API creates a PREPARE PS header and transmits it by issuing a SEND_DATA or MC_SEND_DATA verb. For a mapped conversation, the data_type field of the MC_SEND_DATA VCB must be set to AP_PS_HEADER.
5. On completion of the SEND_DATA or MC_SEND_DATA verb, the vendor API issues a RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb.
6. The RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb returns with a return code of AP_BACKED_OUT, indicating that the remote transaction program issued a BACKOUT verb.
7. The vendor API issues another RECEIVE_AND_WAIT or MC_RECEIVE_AND_WAIT verb to receive the Confirm indication.
8. When the verb completes with the what_rcvd field of the VCB set to AP_CONFIRM, the vendor API issues a CONFIRMED or MC_CONFIRMED verb to acknowledge the BACKOUT verb.
9. The SYNCPT verb is returned to the transaction program with a BACKED_OUT return code when the CONFIRMED or MC_CONFIRMED verb completes.

Windows CSV Overview

Common service verbs (CSVs) are a set of programming functions provided by Microsoft® Host Integration Server 2000 and Microsoft® SNA Server. The CSVs provide convert, log, trace, and transfer services to applications.

The CSVs and information presented in this guide represent an evolving CSV API that is composed of IBM Extended Services for OS/2 version 1.0 and a set of Windows extensions that allow for registering and deregistering the application and that provide an asynchronous entry point for TRANSFER_MS_DATA.

This section describes the verbs available to you and explains how to use them with your applications. A detailed description of each verb is provided in the reference portion of the SDK.

The CSVs are as follows:

CONVERT
: Converts a character string from ASCII to EBCDIC or from EBCDIC to ASCII.

COPY_TRACE_TO_FILE
: Concatenates the contents of the individual application programming interface (API)/link service trace files to form a single trace file.

DEFINE_TRACE
: Enables or disables tracing for specific APIs.

GET_CP_CONVERT_TABLE
: Creates and returns a 256-byte conversion table to translate character strings from a source code page to a target code page.

LOG_MESSAGE
: For OS/2 only, takes a message from a message file, adds specified data to it, and records the message in the error log file. This verb optionally displays the message on the user's screen.

TRANSFER_MS_DATA
: Builds a Systems Network Architecture (SNA) request unit (RU) containing Network Management Vector Transport (NMVT) data. The verb can send the NMVT data to NetView for centralized problem diagnosis and resolution. The data is optionally logged in the event log for Microsoft® Windows 2000, Windows NT®, Windows® 98, Windows® 95, Windows® version 3.*x*, or MS-DOS®, and in the local audit log file for OS/2.

Host Integration Server and SNA Server Asynchronous Support

Asynchronous call completion returns the initial call immediately so the application can continue with other processes. An application that issues a call and does not regain control until the operation completes is not able to perform any other operations. This type of operation, referred to as blocking, is not suited to a server application designed to handle multiple requests from many clients.

Through **RegisterWindowsMessage** with "WinAsyncCSV" as the string, you pass a window handle by which you will be notified of call completion. You then make your call and when it completes, a message is posted to the window handle that you passed, notifying you that the call is complete.

Before Using Windows CSV

The following Windows extensions are of particular importance and should be reviewed before using Windows CSV:

WinAsyncCSV

Provides an asynchronous entry point for TRANSFER_MS_DATA only. If used for any other verb, the behavior will be synchronous. Use this extension instead of the blocking version of the verb if you run your application under Microsoft® Windows® version 3.*x*.

When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinAsyncCSV" as the input string. The *wParam* argument contains the asynchronous task handle returned by the original function call. The *lParam* argument contains the original verb control block (VCB) pointer and can be dereferenced to determine the final return code.

If the function returns successfully, a "WinAsyncCSV" message is posted to the application when the operation completes or the conversation is canceled.

WinCSVCleanup

Terminates and deregisters an application from a Windows CSV implementation.

> An application must call this function to deregister itself from the Windows CSV implementation.

WinCSVStartup

Allows an application to specify the version of Windows CSV required and to retrieve details of the specific CSV implementation.

> An application must call this function to register itself with a Windows CSV implementation before issuing any further Windows CSV calls.

Creating Specific NetView User Alerts

You can create NetView user alerts for users to send. Users identify the alerts by number; each number corresponds to a specific collection of information or requests that the user wants to send via NetView to a host operator.

Host Integration Server and SNA Server leave blank fields for the user alert information in the structure that is returned from the sepdcrec function. To create specific user alerts, create appropriate data structures and call the TRANSFER_MS_DATA verb to send the user alert to NetView.

Using CSVs in C Programs

CSVs are available to applications written in Microsoft C version 5.1 or later. A program written in C calls CSVs through the external function **WINCSV**.

Sample Programs

A collection of sample programs is delivered with the Host Integration Server Software Development Kit and the SNA Server Software Development Kit in the \SDK\SAMPLES directory on the SNA Server CD. For more information, see Sample APPC TPs in the SDK.

CSV Verb Control Block

The only parameter passed to the **WINCSV** function is the address of a VCB. The VCB is a structure made up of variables that identify the verb to be executed, supply information to be used by the verb, and contain information returned by the verb when execution is complete. Each verb has its own VCB structure, which is declared in the file WINCSV.H.

Bit Ordering

Bit 0 refers to the high-order bit in a byte or word. To set bit 0 on in a byte, use the bitwise **OR** operation (=) with a value of 128.

WINCSV Definition

The prototype definition of the **WINCSV** function is as follows:

```
extern void WINAPI WINCSV (LPCSV);
```

The VCB address parameter, a 32-bit pointer, is declared as a long integer and requires casting from a pointer to a long integer.

WINCSV.H File

Use the **#include** command to include the WINCSV.H file in any application that issues CSVs.

The WINCSV.H file, which is included with the Host Integration Server and SNA Server Software Development Kits, contains:

- The CSV function prototype.
- The structure declarations for the CSV VCBs.
- The **#define** statements that substitute meaningful symbolic constants for hexadecimal values supplied to and returned by CSVs.

If a **#define** statement pertains to a hexadecimal value that is longer than one byte, a comment shows how the hexadecimal value is stored in memory.

When setting or testing CSV parameters, use the symbolic constants defined by the WINCSV.H file. When examining trace files or the contents of memory, use the hexadecimal values.

Issuing a CSV

The procedure for issuing a CSV is shown in the following sample code that uses CONVERT.

To issue a CSV

1. Create a structure variable from the VCB structure that applies to the verb to be issued.

```
#include <wincsv.h>
    .
    .
struct convert  conv_block;
```

   The VCB structures are declared in the WINCSV.H file; one of these structures is named **CONVERT**.

2. Clear (set to zero) the variables within the structure.

```
memset( conv_block, '\0', sizeof( conv_block ) );
```

   This procedure is not required. However, it helps in debugging and reading the contents of memory. It also eliminates the possibility that future versions of a verb are sensitive to fields that are ignored in the current version.

3. Assign values to the required VCB variables.

```
conv_block.opcode = SV_CONVERT;
conv_block.direction = SV_ASCII_TO_EBCDIC;
conv_block.char_set = SV_AE;
conv_block.len = sizeof(tpstart_name);
conv_block.source = (LPBYTE) tpstart_name;
conv_block.target = (LPBYTE) tpstart.tp_name;
```

   The values SV_CONVERT, SV_ASCII_TO_EBCDIC, and SV_AE are symbolic constants representing integers. These constants are defined in the WINCSV.H file.

   The character array TPSTART_NAME contains an ASCII string to be converted to EBCDIC and placed in the character array TPSTART.TP_NAME.

4. Invoke the verb. The only parameter is a pointer to the address of the structure containing the VCB for the verb.

```
ACSSVC((LONG) &conv_block);
```

   You can also use the following statement:

```
ACSSVC_C((LONG) &conv_block);
```

5. Use the values returned by the verb.

```
if( conv_block.primary_rc == SV_OK ) {
/* other statements */
    .
    .
    .
```

Support for APPC Automatic Logon

This section describes the support for automatic logon for APPC applications available in Microsoft® Host Integration Server 2000 and Microsoft® SNA Server version 3.0 with Service Pack 1 or higher. This feature requires specific configuration by the network administrator: For more information on configuring this feature, see the SNA Server online documentation.

The APPC application must be invoked on the LAN side from a client of Host Integration Server or SNA Server. The client must be logged into a Windows 2000 or Windows NT® domain, but the client can be running on any operating system supported by the Host Integration Server or SNA Server APPC APIs.

The client application is coded to use "program" level security, with a special hard-coded APPC user name MS$SAME and password MS$SAME. When this session allocation flows from client to Host Integration Server or SNA Server, the server looks up the host account and password corresponding to the Windows 2000 or Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

To use this feature for an APPC application, the **user_id** and **pwd** fields in the ALLOCATE or MC_ALLOCATE verbs must be hard-coded to this special string and **security** must be set to AP_PGM.

APPC Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about the verbs, extensions, and return codes that make up the APPC programming interface.

This section contains:

APPC Management Verbs

This section describes the APPC management verbs:

ACTIVATE_SESSION

CNOS

DEACTIVATE_SESSION

DISPLAY

The management verbs allow you to establish APPC LU 6.2 session limits, obtain configuration information and current operating values for the SNA node, and activate or deactivate sessions. The description of each verb provides:

- A definition of the verb.
- The structure defining the VCB used by the verb. The structure is contained in the WINAPPC.H file. The length of each VCB field is in bytes. Fields beginning with **reserv** (for example, **reserv2**) are reserved.
- The parameters (VCB fields) supplied to and returned by APPC. A description of each parameter is provided, along with its possible values and other information.
- The conversation state(s) in which the verb can be issued.
- The state(s) to which the conversation can change upon return from the verb. Conditions that do not cause a state change are not noted. For example, parameter checks and state checks do not cause a state change.
- Additional information describing the verb.

Most parameters supplied to and returned by APPC are hexadecimal values. To simplify coding, these values are represented by meaningful symbolic constants, which are established by **#define** statements in the WINAPPC.H header file. For example, the **opcode** (operation code) member of the **mc_send_data** structure used by the **MC_SEND_DATA** verb is the hexadecimal value represented by the symbolic constant AP_M_SEND_DATA. Use only the symbolic constants when writing TPs.

ACTIVATE_SESSION

The **ACTIVATE_SESSION** verb requests Host Integration Server or SNA Server to activate a session between the local LU and a specified partner LU, using a specified mode. This verb completes either when the specified session has become active or when it has failed.

The following structure describes the verb control block used by the **ACTIVATE_SESSION** verb.

```
typedef struct activate_session {
    unsigned short  opcode;
    unsigned char   reserv2[2];
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   reserv3[8];
    unsigned char   lu_alias[8];
    unsigned char   plu_alias[8];
    unsigned char   mode_name[8];
    unsigned char   fqplu_name[17];
    unsigned char   polarity;
    unsigned char   session_id[8];
    unsigned long   conv_group_id;
    unsigned char   reserv4[1];
    unsigned char   type;
    HANDLE          deactivation_event;
    unsigned short* p_deactivation_status;
    unsigned char   reserv5[10];
} ACTIVATE_SESSION;
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_ACTIVATE_SESSION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

reserv3
A reserved field.

lu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the LU. If the default local LU is to be used, fill this parameter with spaces.

plu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the partner LU. If the default remote LU is to be used, fill this parameter with spaces. If the partner LU is to be specified with the fqplu_name parameter, fill this parameter with binary zeros.

mode_name
Supplied parameter. Specifies the EBCDIC (type A) mode name.

fqplu_name
Supplied parameter. Provides the partner LU name in EBCDIC (type A) when no **plu_alias** name is defined at the local node and the partner LU is located at a different node. This parameter is ignored if plu_alias is specified.

polarity
Supplied parameter. Specifies the polarity for the session. The possible values are AP_POL_EITHER, AP_POL_FIRST_SPEAKER, and AP_POL_BIDDER.

If AP_POL_EITHER is set, ACTIVATE_SESSION activates a first speaker session if available, otherwise a bidder session is activated.

If AP_POL_FIRST_SPEAKER is set, ACTIVATE_SESSION only succeeds if a session of the requested polarity is available.

If AP_POL_BIDDER is set, ACTIVATE_SESSION only succeeds if a session of the requested polarity is available.

session_id
Returned parameter. Provides the 8-byte identifier of the activate session.

conv_group_id
Returned parameter. Provides the conversation group identifier. This parameter can be specified on ALLOCATE and MC_ALLOCATE verbs to start conversations on this particular session.

reserv4
A reserved field.

type
Supplied parameter. Specifies the type of activation. Possible values are AP_ACT_ACTIVE and AP_ACT_PASSIVE.

If AP_ACT_ACTIVE is specified, then Host Integration Server or SNA Server will attempt to start the required session (by sending the BIND or INIT-SELF).

If AP_ACT_PASSIVE is specified, then Host Integration Server or SNA Server will not attempt to start the session and the verb will complete when the partner has started the session.

deactivation_event
Supplied parameter. Provides an event handle that APPC is to signal when the session is deactivated. The event handle should be obtained by calling either the **CreateEvent** or **OpenEvent** Win32® function.

p_deactivation_status
Returned parameter. A pointer to a value that is set when the deactivation event is signaled to provide completion status. The following values can be returned.
AP_SESSION_DEACTIVATED
AP_COMM_SUBSYSTEM_ABENDED

reserv5
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully. The secondary return code indicates the polarity of the established session. The following values can be returned.

AP_POL_FIRST_SPEAKER

AP_POL_BIDDER

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_INVALID_LU_ALIAS

Secondary return code; APPC cannot find the specified **lu_alias** among those defined.

AP_INVALID_PLU_ALIAS

Secondary return code; APPC did not recognize the specified **plu_alias**.

AP_INVALID_MODE_NAME

Secondary return code; APPC did not recognize the specified **mode_name**.

AP_INVALID_FQPLU_NAME

Secondary return code; APPC did not recognize the specified **fqplu_name**.

AP_INVALID_POLARITY

Secondary return code; APPC did not recognize the specified **polarity**.

AP_INVALID_TYPE

Secondary return code; APPC did not recognize the specified **type**.

AP_ACTIVATION_FAIL_NO_RETRY
Primary return code; the session could not be activated because of a condition that requires action (such as a configuration mismatch or a session protocol error).

AP_ACTIVATION_FAIL_RETRY
Primary return code; the session could not be activated because of a temporary condition (such as a link failure).

AP_SESSION_LIMITS_EXCEEDED
Primary return code; the session could not be activated because the session limits have been exceeded.

AP_SESSION_LIMITS_CLOSED
Primary return code; the session could not be activated because the session limits are closed (i.e. zero).
AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions occurred:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (local area network error occurred).
- The SnaBase at the TP's computer encountered an ABEND.
- The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

Remarks

This verb is supported using APPC only on Windows 2000, Windows NT, Windows 98, and Windows 95.

This verb supports both active and passive activation.

The active form of this verb results in Host Integration Server or SNA Server trying to initiate the session (by sending a BIND for independent LUs or an INIT-SELF for dependent LUs). The active form of this verb will also result in the following behavior.

- If the connection to the partner LU is inactive and is configured as on-demand, the Node will attempt to start the connection.
- If dynamic partnering is being used, the Node will set up the LU-LU/MODE partnership.
- If CNOS has not run, the Node will start CNOS (but will not change any of the session limits).

The passive form does not attempt to start the session, but completes when the LU is started by a BIND from its partner LU. For independent LUs, multiple passive ACTIVATE_SESSION verbs can be queued up for the same LU-LU/MODE, and complete in turn as new sessions are started.

This verb also includes a deactivation event, which is posted when the session is deactivated by any method other than a DEACTIVATE_SESSION verb (for example, an unsolicited UNBIND from its partner LU results in this event being posted).

CNOS

The **CNOS** (Change Number of Sessions) verb establishes APPC LU 6.2 session limits.

The following structure describes the verb control block used by the **CNOS** verb.

```
typedef struct cnos {
    unsigned short  opcode;
    unsigned char   reserv2[2];
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   key[8];
    unsigned char   lu_alias[8];
    unsigned char   plu_alias[8];
    unsigned char   fqplu_name[17];
    unsigned char   reserv3;
    unsigned char   mode_name[8];
    unsigned int    mode_name_select:1;
    unsigned int    set_negotiable:1;
    unsigned int    reserv4:6;
    unsigned int    reserv5:8;
    unsigned short  plu_mode_sess_lim;
    unsigned short  min_conwinners_source;
    unsigned short  min_conwinners_target;
    unsigned short  auto_act;
    unsigned int    drain_target:1;
    unsigned int    drain_source:1;
    unsigned int    responsible:1;
    unsigned int    reserv6:5;
    unsigned int    reserv7:8;
} CNOS;
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_CNOS.

reserv2
A reserved field.›

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

key
Supplied parameter. Specifies either the master or service key in ASCII, if the keylock feature has been secured.

lu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the LU.

plu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the partner LU.

fqplu_name
Supplied parameter. Provides the partner LU name in EBCDIC (type A) when no **plu_alias** name is defined at the local node and the partner LU is located at a different node.

mode_name
Supplied parameter. Specifies the EBCDIC (type A) mode name to be used when the value of **mode_name_select** is AP_ONE.

mode_name_select
Supplied parameter. Specifies the mode name select for which your program is setting or resetting the session limits and contention-winner polarities. Allowed values are AP_ALL or AP_ONE.

set_negotiable
Supplied parameter. Specifies whether APPC is to change the current setting for the maximum negotiable session limit. Allowed values are AP_YES and AP_NO.

reserv4
A 6-bit reserved field.

reserv5
An 8-bit reserved field.›
plu_mode_sess_lim
Supplied parameter. Specifies the session limit when the value for **set_negotiable** is YES. Allowed values are 0 to 32767.
min_conwinners_source
Supplied parameter. Specifies the number of sessions of which the LU is guaranteed to be the contention winner. Allowed values are 0 to 32767.
min_conwinners_target
Supplied parameter. Specifies the minimum number of sessions of which the target LU is guaranteed to be the contention winner. Allowed values are 0 to 32767.
auto_act
Supplied parameter. Specifies the number of the local LU's contention-winner sessions for APPC to activate automatically. Allowed values are 0 to 32767. See the Remarks section of this topic before using this parameter.
drain_target
Supplied parameter. Specifies whether the target LU can drain its waiting (outbound) allocation requests. Allowed values are AP_YES and AP_NO.
drain_source
Supplied parameter. Specifies whether the source LU can drain its waiting (outbound) allocation requests. Allowed values are AP_YES and AP_NO.
responsible
Supplied parameter. Specifies which LU is responsible for deactivating the sessions as a result of resetting the session limit for parallel-session connections. Allowed values are AP_SOURCE and AP_TARGET.
reserv6
A 5-bit reserved field.
reserv7
An 8-bit reserved field.›

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_CNOS_ACCEPTED

Secondary return code; APPC accepts the session limits and responsibility as specified.

AP_CNOS_NEGOTIATED

Secondary return code; APPC accepts the session limits and responsibility as negotiable by the partner LU. Values that can be negotiated are:

plu_mode_session_limit

min_conwinners_source

min_conwinners_target

responsible

drain_target

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE or MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CNOS_LOCAL_RACE_REJECT
Primary return code; APPC is currently processing a **CNOS** verb issued by a local LU.
AP_CNOS_PARTNER_LU_REJECT

Primary return code; the partner LU rejected a **CNOS** request from the local LU.

AP_CNOS_MODE_CLOSED

Secondary return code; the local LU cannot negotiate a nonzero session limit because the local maximum session limit at the partner LU is zero.

AP_CNOS_MODE_NAME_REJECT

Secondary return code; the partner LU does not recognize the specified mode name.

AP_CNOS_COMMAND_RACE_REJECT

Secondary return code; the local LU is currently processing a **CNOS** verb issued by the partner LU.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a local area network error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE or MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** or **MC_ALLOCATE** request.

When **ALLOCATE** or **MC_ALLOCATE** produces this return code for an Host Integration Server or SNA Server system, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_KEY
Primary return code; the supplied key was incorrect.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_ALL_MODE_MUST_RESET

Secondary return code; APPC does not permit a nonzero session limit when the **mode_name_select** parameter indicates AP_ALL.

AP_AUTOACT_EXCEEDS_SESSLIM

Secondary return code; on the **CNOS** verb, the value for **auto_act** is greater than the value for **plu_mode_sess_lim**.

AP_BAD_LU_ALIAS

Secondary return code; APPC cannot find the specified **lu_alias** among those defined.

AP_BAD_PARTNER_LU_ALIAS

Secondary return code; APPC did not recognize the supplied **plu_alias**.

AP_BAD_SNASVCMG_LIMITS

Secondary return code; your program specified invalid settings for **plu_mode_sess_lim**, **min_conwinners_source**, or **min_conwinners_target** when **mode_name** was supplied.

AP_CHANGE_SRC_DRAINS

Secondary return code; APPC does not permit **mode_name_select** (ONE) and **drain_source** (YES) when **drain_source** (NO) is currently in effect for the specified mode.

AP_CNOS_IMPLICIT_PARALLEL

Secondary return code; APPC does not permit a program to change the session limit for a mode other than the SNASVCMG mode for the implicit partner template when the template specifies parallel sessions. (The term "template" is used because many of the actual values are yet to be filled in.)

AP_CPSVCMG_MODE_NOT_ALLOWED

Secondary return code; the mode named CPSVCMG cannot be specified as the **mode_name** on the deactivate session verb.

AP_EXCEEDS_MAX_ALLOWED

Secondary return code; your program issued a **CNOS** verb, specifying a **plu_mode_sess_lim** number and **set_negotiable** (AP_NO).

AP_MIN_GT_TOTAL

Secondary return code; the sum of **min_conwinners_source** and **min_conwinners_target** specifies a number greater than **plu_mode_sess_lim**.

AP_MODE_CLOSED

Secondary return code; the local LU cannot negotiate a nonzero session limit because the local maximum session limit at the partner LU is zero.

AP_RESET_SNA_DRAINS

Secondary return code; SNASVCMG does not support the drain parameter values.

AP_SINGLE_NOT_SRC_RESP

Secondary return code; for a single-session **CNOS** verb, APPC permits only the local (source) LU to be responsible for deactivating sessions.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_CANT_RAISE_LIMITS

Secondary return code; APPC does not permit setting session limits to a nonzero value unless the limits currently are zero.

AP_LU_DETACHED

Secondary return code; a command has reset the definition of the local LU before **CNOS** tried to specify the LU.

AP_SNASVCMG_RESET_NOT_ALLOWED

Secondary return code; your local program attempted to issue the **CNOS** verb for the mode named SNASVCMG, specifying a session limit of zero.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC verb from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

CNOS identifies an LU by alias alone. If the same local LU alias is used multiple times in a domain (for backup or other purposes) and that LU alias is specified through **CNOS**, the verb can flow to a different LU than the one intended.

If **CNOS** is not issued to set the mode session limit before a program issues its first APPC ALLOCATE, MC_ALLOCATE, SEND_CONVERSATION, or MC_SEND_CONVERSATION, or Common Programming Interface for Communications (CPI-C)

Allocate call for a given partner LU and mode, APPC will internally generate a session limit using the value from the mode definition.

When setting the limits for a parallel-session connection, the two LUs negotiate the mode session limits, drain settings, and responsibility values. APPC updates these parameters in **CNOS** to reflect the settings agreed to by both LUs during negotiation. Your program can issue DISPLAY to obtain the negotiated values for the mode session limit.

No **CNOS** negotiation occurs when setting the limits for a single session (that is, the two LUs do not negotiate drain settings or responsibility values). Therefore, coordinate the mode definition parameter settings between partner LUs using a single-session connection by defining a single session mode at each node.

As part of setting up the initial limits, **CNOS** also sets the guaranteed (that is, the minimum) number of contention-winner and contention-loser sessions and sets the automatic activation count for the source LU's contention-winner sessions. The action of **CNOS** normally affects only the group of sessions with the specified mode name between the source LU and the target LU. Alternatively, one **CNOS** can reset the session limits of all modes for a partner LU.

APPC enforces the new mode session limit and the contention-winner polarities until one side or the other changes them by issuing a subsequent **CNOS** verb. The **CNOS** transaction is invisible at the target LU's API, regardless of which LU is the target. The results of the **CNOS** transaction can be obtained using **DISPLAY**.

Setting a Session Limit to Zero

After **CNOS** raises the session limit above zero, it can reset the limit to zero only. It cannot set the session limit to a value that is not zero, and it cannot redistribute the number of sessions allocated as the contention winners and losers. Therefore, your program cannot change the mode session limits if the two LUs have already set the limits to a nonzero value, regardless of which LU initiated the **CNOS** transaction.

A program can change the session limits from a nonzero value, as long as the program first changes the session limit to zero. For example, if the session limit is 8, a program can change it to 6 by first issuing **CNOS** and changing the session limit to zero, and then issuing **CNOS** again and setting the session limit to 6.

APPC can activate one or more LU-LU sessions with the specified mode name as a result of initializing the session limit. You cannot use **CNOS** to activate sessions between two LUs on the same server. APPC deactivates all LU-LU sessions for the specified mode name (or for all mode names for a partner LU) as a result of resetting the session limit to zero. APPC deactivates each session as it becomes free and does not interrupt active conversations.

A separate value, the maximum negotiable session limit, is used in **CNOS** negotiations. If the **set_negotiable** value is AP_YES, the mode session limit value given in this **CNOS** verb also sets the maximum negotiable session limit.

The **lu_alias** and **plu_alias** parameters are 8-byte ASCII character strings. If the name is fewer than eight bytes, it must be padded on the right with ASCII spaces.

You can specify the SNA-defined mode name SNASVCMG for **mode_name**. Use this mode only in a **CNOS** transaction when the source LU and the target LU use parallel user sessions. However, when resetting the session limits to zero for the SNASVCMG PU 2.1 node, the session limits of all other modes between the two LUs must be reset first. The PU 2.1 mode name is a type A EBCDIC character string. A mode name consisting of all spaces is supported. The SNA-defined mode name CPSVCMG is not allowed.

When specifying **plu_mode_sess_lim**, if the mode session limit is currently greater than zero, the value of this parameter must be zero. **CNOS** can raise the limit above zero, but the next **CNOS** must set the value to zero. A single **CNOS** cannot change the mode session limit from one nonzero number to another.

When raising the mode session limit above zero for a parallel-session connection, the target LU can negotiate its parameter to a value greater than zero and less than the specified session limit. The specified or negotiated limit then becomes the new mode session limit and is returned in this field.

The value specified for this parameter must be greater than or equal to the sum of the values specified in the **CNOS** **min_conwinners_source** and **min_conwinners_target** parameters.

Do not reset the SNASVCMG session limit to zero until all other mode session limits between the two LUs are reset to zero and the count of active sessions for all modes (except SNASVCMG) for the partner LU is zero.

The mode session limit should be large enough to accommodate all active conversations in the mode for all TPs.

For **min_conwinners_source** and **min_conwinners_target**, the sum of both parameters cannot exceed the mode session limit. For single-session connections, these parameters specify the desired contention-winner sessions for the target and source LUs. For the SNASVCMG mode name (with a mode session limit of 2 or 1), the specified minimum number of contention-winner sessions for the target LU must be 1. For the source LU, with a mode session limit of 2, the number must be 1; with a mode session limit of 1, the number must be 0. APPC uses these parameters only when the mode session limit is set to a nonzero value.

APPC uses **auto_act** only when the mode session limit is set to a nonzero value. If the value is greater than the **min_conwinners_source** value, APPC uses the new minimum number of contention winners for the source LU as the autoactivation limit.

> The **auto_act** parameter can conflict with the on-demand definition of a connection. Autoactivations by either peer partner can re-establish sessions and connections, possibly resulting in a thrashing situation. Therefore, avoid specifying autoactivation between peer PU 2.1 nodes using on-demand connections.

Whether an LU deactivates a session immediately after the current conversation or after all queued conversations are complete depends on the **drain_source** and **drain_target** parameters.

If an LU is to drain its waiting (outbound) allocation requests, it continues to allocate conversations to active sessions. The responsible LU deactivates a session only when the conversation allocated to the session is deallocated and no request is waiting for allocation to any session with the specified mode name between the two LUs. The allocation of a waiting request takes precedence over the deactivation of a session.

If an LU is not to drain its waiting (outbound) allocation requests, the responsible LU deactivates a session as soon as the conversation allocated to the session is deallocated. If no conversation is allocated to the session, the responsible LU deactivates

the session immediately. However, this verb does not force deallocation of active conversations.

The **responsible** and **mode_name_select** parameters are interrelated as follows:

- APPC ignores the **responsible** parameter for mode names for which the session limit is currently zero if this **CNOS** verb specifies **mode_name_select** (AP_ALL).
- If **CNOS** specifies **mode_name_select** (AP_ONE) with a mode session limit of zero, and the current session limit for that mode name is already zero, the **responsible** parameter must specify the same LU (SOURCE or TARGET) that is currently responsible for deactivating sessions. APPC uses this parameter only when **CNOS** specifies a mode session limit of zero.

For parallel-session connections, the **drain_source** and **mode_name_select** parameters are interrelated as follows:

- If **CNOS** specifies **mode_name_select** (AP_ALL) and **drain_source** (AP_YES), APPC ignores **drain_source** for those mode names for which the session limit is currently zero.
- If **CNOS** specifies **mode_name_select** (AP_ALL) and **drain_source** (AP_NO), APPC accepts **drain_source** for all mode names. APPC ends draining for any mode currently draining its requests.
- If **CNOS** specifies **mode_name_select** (AP_ONE), and **drain_source** (AP_YES) is currently in effect, **drain_source** (AP_NO) directs APPC to end the draining at the source LU for requests for the specified mode name.
- If **CNOS** specifies **mode_name_select** (AP_ONE) and **drain_source** (AP_NO) is currently in effect, your program must specify **drain_source** (AP_NO) again.

For parallel-session connections, the **drain_target** parameter and the **mode_name_select** parameter are interrelated as follows:

- If **CNOS** specifies **mode_name_select** (AP_ALL) and **drain_target** (AP_YES), APPC ignores **drain_target** for the mode names for which the session limit is currently zero.
- If **CNOS** specifies **mode_name_select** (AP_ALL) and **drain_target** (AP_NO), APPC accepts **drain_target** for all mode names, regardless of the current session limit. Any draining of waiting (outbound) allocation requests at the target LU is ended.
- If **CNOS** specifies **mode_name_select** (AP_ONE) and **drain_target** (AP_YES) is currently in effect, **drain_target** (AP_NO) ends the target LU's draining.
- If **CNOS** specifies **mode_name_select** (AP_ONE) and **drain_target** (AP_YES), and **drain_target** (AP_NO) is currently in effect, the target LU can either accept **drain_target** (AP_YES) or negotiate the parameter to AP_NO. After the target LU accepts the **drain_target** (AP_YES) parameter, it can drain any remaining waiting (outbound) allocation requests.

DEACTIVATE_SESSION

The **DEACTIVATE_SESSION** verb requests Host Integration Server or SNA Server to deactivate a particular session between the local LU and a specified partner LU, or all sessions on a particular mode.

The following structure describes the verb control block used by the **DEACTIVATE_SESSION** verb.

```
typedef struct deactivate_session {
    unsigned short  opcode;
    unsigned char   reserv2[2];
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   reserv3[8];
    unsigned char   lu_alias[8];
    unsigned char   session_id[8];
    unsigned char   plu_alias[8];
    unsigned char   mode_name[8];
    unsigned char   type;
    unsigned char   reserv4[3];
    unsigned short  sense_data;
    unsigned char   fqplu_name[17];
    unsigned char   reserv5[19];
} DEACTIVATE_SESSION;
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_DEACTIVATE_SESSION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

reserv3
A reserved field.

lu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the LU.

session_id
Supplied parameter. Provides the 8-byte identifier of the session to deactivate (returned on the ACTIVATE_SESSION verb). If this field is set to 8 binary zeros, Host Integration Server or SNA Server deactivates all sessions for the partner LU and mode.

plu_alias
Supplied parameter. Provides the 8-byte ASCII name used locally for the partner LU. If the default remote LU is to be used, fill this parameter with spaces. If the partner LU is to be specified with the fqplu_name parameter, fill this parameter with binary zeros.

mode_name
Supplied parameter. Specifies the EBCDIC (type A) mode name.

type
Supplied parameter. Specifies the type of deactivation. Possible values are:

AP_DEACT_CLEANUP

Deactivate the session immediately, without waiting for sessions to end.

AP_DEACT_NORMAL

Do not deactivate the session until all conversations using the session have ended.

sense_data
Returned parameter. Specifies the deactivation sense data for the session.

reserv4
A reserved field.

fqplu_name

Supplied parameter. Provides the partner LU name in EBCDIC (type A) when no **plu_alias** name is defined at the local node and the partner LU is located at a different node. This parameter is ignored if plu_alias is specified.

reserv5

A reserved field.

Return Codes

AP_OK

Primary return code; the verb executed successfully. The secondary return code indicates the polarity of the established session. The following values can be returned.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error, specified by one of the following secondary return codes:

AP_INVALID_LU_ALIAS

Secondary return code; APPC cannot find the specified **lu_alias** among those defined.

AP_INVALID_PLU_ALIAS

Secondary return code; APPC did not recognize the specified **plu_alias**.

AP_INVALID_SESSION_ID

Secondary return code; APPC did not recognize the specified **session_id**.

AP_INVALID_MODE_NAME

Secondary return code; APPC did not recognize the specified **mode_name**.

AP_INVALID_FQPLU_NAME

Secondary return code; APPC did not recognize the specified **fqplu_name**.

AP_INVALID_TYPE

Secondary return code; APPC did not recognize the specified **type**.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions occurred:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a local area network error occurred).
- The SnaBase at the TP's computer encountered an ABEND.
- The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_THREAD_BLOCKING

Primary return code; the calling thread is already in a blocking call.

Remarks

This verb is supported using APPC only on Windows 2000, Windows NT, Windows 98, and Windows 95.

DISPLAY

The **DISPLAY** verb returns configuration information and current operating values for the SNA node.

It is recommended that you use the GetAppcConfig Windows extension function to obtain system configuration information relating to APPC LUs. Users of 5250 emulators, in particular, should use the **GetAPPCConfig** Windows extension.

> **Note** Because of the nature of client/server architecture, the implementation of the **DISPLAY** on Host Integration Server 2000 and SNA Server contains important differences from the IBM Extended Services for OS/2 version 1.0 (IBM ES for OS/2 version 1.0) on which it was based. These differences are described in the following topics. For information on configuring **DISPLAY** in Host Integration Server 2000 Manager, see the Microsoft Host Integration Server 2000 online books.

> **Note** For applications that use the APPC **DISPLAY** verb in IBM ES for OS/2 version 1.0 compatibility mode and that do not use the Host Integration Server or SNA Server extensions for enumerating all active servers and connections, Host Integration Server and SNA Server will randomly choose a default **DISPLAY** connection, unless a specific default **DISPLAY** connection has been configured in Host Integration Server Manager or SNA Explorer. This connection is used as the basis for all **DISPLAY** requests. For information about specifying the default **DISPLAY** connection, see the Microsoft Host Integration Server 2000 online books.

The following structure describes the verb control block used by the **DISPLAY** verb.

```
struct display {
    unsigned short  opcode;
    unsigned char   reserv2[2];
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned long   init_sect_len;
    unsigned long   buffer_len;
    unsigned char FAR * buffer_ptr;
    unsigned long  num_sections;
    unsigned long  display_len;
    unsigned long  area_needed;
    unsigned char  sna_global_info;
    unsigned char  lu62_info;
    unsigned char  am_info;
    unsigned char  tp_info;
    unsigned char  sess_info;
    unsigned char  link_info;
    unsigned char  lu_0_3_info;
    unsigned char  gw_info;
    unsigned char  x25_physical_link_info;
    unsigned char  sys_def_info;
    unsigned char  adapter_info;
    unsigned char  lu_def_info;
    unsigned char  plu_def_info;
    unsigned char  mode_def_info;
    unsigned char  link_def_info;
    unsigned char  ms_info;
    struct sna_global_info_sect FAR * sna_global_info_ptr;
    struct lu62_info_sect FAR * lu62_info_ptr;
    struct am_info_sect FAR * am_info_ptr;
    struct tp_info_sect FAR * tp_info_ptr;
    struct sess_info_sect FAR * sess_info_ptr;
    struct link_info_sect FAR * link_info_ptr;
    struct lu_0_3_info_sect FAR * lu_0_3_info_ptr;
    struct gw_info_sect FAR * gw_info_ptr;
    struct x25_physical_link_info_sect FAR * x25_physical_link_info_ptr;
    struct sys_def_info_sect FAR * sys_def_info_ptr;
    struct adapter_info_sect FAR * adapter_info_ptr;
    struct lu_def_info_sect FAR * lu_def_info_ptr;
    struct plu_def_info_sect FAR * plu_def_info_ptr;
    struct mode_def_info_sect FAR * mode_def_info_ptr;
    struct link_def_info_sect FAR * link_def_info_ptr;
    struct ms_info_sect FAR * ms_info_ptr;
} DISPLAY;
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_DISPLAY.

reserv2
A reserved field, this value must be set to NULL.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

init_sect_len
Supplied parameter. Specifies the number of bytes in the initial section of the VCB, up to the beginning of information pointers. This parameter and the **num_sections** parameter must be set to specific values depending on the format being requested. See the notes below for details.

buffer_len
Supplied parameter. Specifies the length (0 to 65535 bytes) of the passed display data buffer.

buffer_ptr
Supplied parameter. Provides the address of the display data buffer that will contain the requested information.

num_sections
Supplied parameter. Specifies the maximum number of information sections that can be returned by the verb. This parameter and the **init_sect_len** parameter must be set to specific values depending on the format being requested. See the notes below for details.

display_len
Returned parameter. Provides the total number of bytes used that are returned into the display data buffer.

area_needed
Returned parameter. Provides the total number of bytes needed for all of the displayed data.

sna_global_info
Supplied parameter. Specifies if global information is requested. Allowed values are AP_YES and AP_NO.

lu62_info
Supplied parameter. Specifies if information on all active LUs, their partners, and their modes is requested. Allowed values are AP_YES and AP_NO.

am_info
Supplied parameter. Specifies if Attach Manager information on the defined TP is requested. Allowed values are AP_YES and AP_NO.

> **Note** This option is not supported by Host Integration Server or SNA Server and this parameter must be set to AP_NO.

tp_info
Supplied parameter. Specifies if information on the active TPs and any active conversations is requested. Allowed values are AP_YES and AP_NO.

> **Note** This option is not supported by Host Integration Server or SNA Server and this parameter must be set to AP_NO.

sess_info
Supplied parameter. Specifies if information on sessions is requested. Allowed values are AP_YES and AP_NO.

link_info
Supplied parameter. Specifies if information on the active SNA logical lines is requested. Allowed values are AP_YES and AP_NO.

lu_0_3_info
Supplied parameter. Specifies if information on logical units type 0, 1, 2, and 3 is requested. Allowed values are AP_YES and AP_NO.

gw_info
Supplied parameter. Specifies if information on the SNA gateway is requested. Allowed values are AP_YES and AP_NO.

x25_physical_link_info
Supplied parameter. Specifies if X.25 information is required. Allowed values are AP_YES and AP_NO.

> **Note** This option is not supported by Host Integration Server or SNA Server and this parameter must be set to AP_NO.

sys_def_info
Supplied parameter. Specifies if information about the default LU, node names, and default parameters for inbound and outbound implicit partners is requested. Allowed values are AP_YES and AP_NO.

adapter_info
Supplied parameter. Specifies if information about the configured communications adapters is requested. Allowed values are AP_YES and AP_NO. This parameter must be set to AP_NO when NS/2 format is requested.

lu_def_info
Supplied parameter. Specifies if information about the defined LUs is requested. Allowed values are AP_YES and AP_NO.

plu_def_info
Supplied parameter. Specifies if information about the defined partner LUs is requested. Allowed values are AP_YES and AP_NO.

mode_def_info
Supplied parameter. Specifies if information about the defined nodes is requested. Allowed values are AP_YES and AP_NO.

link_def_info
Supplied parameter. Specifies if information about the defined logical links is requested. Allowed values are AP_YES and AP_NO.

ms_info
Supplied parameter. Specifies if information about management services is requested. Allowed values are AP_YES and AP_NO. This parameter must be set to AP_NO when NS/2 format is requested.

sna_global_info_ptr
Returned parameter. Indicates the address of the beginning of SNA global information in the data buffer.

lu62_info_ptr
Returned parameter. Indicates the address of the beginning of LU 6.2 information in the data buffer.

am_info_ptr
Returned parameter. Indicates the address of the beginning of the Attach Manager information in the data buffer.

> **Note** This option is not supported by Host Integration Server or SNA Server.

tp_info_ptr
Returned parameter. Indicates the address of the beginning of TP information in the data buffer.

> **Note** This option is not supported by Host Integration Server or SNA Server.

sess_info_ptr
Returned parameter. Indicates the address of the beginning of session information in the data buffer.

link_info_ptr
Returned parameter. Indicates the address of the beginning of link information in the data buffer.

lu_0_3_info_ptr
Returned parameter. Indicates the address of the beginning of LU information in the data buffer.

gw_info_ptr
Returned parameter. Indicates the address of the beginning of gateway information in the data buffer.

x25_physical_link_info_ptr
Returned parameter. Indicates the address of the beginning of X.25 information in the data buffer.

> **Note** This option is not supported by Host Integration Server or SNA Server.

sys_def_info_ptr
Returned parameter. Indicates the address of the beginning of system default information in the data buffer.

adapter_info_ptr
Returned parameter. Indicates the address of the beginning of adapter information in the data buffer.

lu_def_info_ptr
Returned parameter. Indicates the address of the beginning of local LU definition information in the data buffer.

plu_def_info_ptr
Returned parameter. Indicates the address of the beginning of partner LU definition information in the data buffer.

mode_def_info_ptr
Returned parameter. Indicates the address of the beginning of mode definition information in the data buffer.

link_def_info_ptr
Returned parameter. Indicates the address of the beginning of link definition information in the data buffer.

ms_info_ptr
Returned parameter. Indicates the address of the beginning of management services information in the data buffer.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_DISPLAY_INVALID_CONSTANT

Secondary return code; the value supplied for NUM_SECTIONS or INIT_SEC_LEN is invalid.

AP_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

AP_DISPLAY_INFO_EXCEEDS_LEN

Secondary return code; the returned **DISPLAY** information did not fit in the buffer.

AP_INVALID_DATA_SEGMENT

Secondary return code; the segment containing the data buffer is too small for the specified data length.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation has encountered an ABEND.
- The connection between the TP and the node type 2.1 has been broken (a LAN error).
- The SnaBase at the TP's computer has encountered an ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

DISPLAY identifies an LU by alias alone. If the same local LU alias is used multiple times in a domain (for backup or other purposes) and that LU alias is specified through **DISPLAY**, the verb can flow to a different LU than the one intended.

For the **DISPLAY** verb to return successfully, a specific connection must be defined in the Host Integration Server 2000 Manager or SNA Explorer program Display Verb dialog box. IBM originally defined the **DISPLAY** verb with the IBM OS/2 Extended Edition product which assumed a single connection. However, because Host Integration Server and SNA Server support multiple connections, the specific connection associated with the **DISPLAY** verb must be configured.

The **DISPLAY** verb requires a user-supplied buffer for the return of system information. If the buffer is not large enough, APPC returns the AP_DISPLAY_INFO_EXCEEDS_LEN return code, along with the size actually needed at the time of the request (in the **area_needed** parameter). One possible strategy for the use of this verb follows:

- If the **buffer_len** value is less than the **area_needed** value returned by APPC, and the required length is less than 64 kilobytes (K), then increase the size of the display buffer to equal or greater than the **area_needed** value.
- If the **area_needed** value is greater than 64K, you can choose to request each information section individually. Or, you can take the following steps:

    1. Process the information sections with complete information, whose total number displayed equals the total actual number.
    2. Choose a subset of the information sections you requested that contains incomplete information, and reissue the verb requesting those information sections.
    3. Repeat steps 1 and 2 as needed.

    Note If an individual information section is greater than 64K, then you cannot get all of the requested information from APPC.

The **DISPLAY** verb should not executed from different threads of the same process, since it is not thread-safe.

The **DISPLAY** verb returns AP_DISPLAY_INVALID_CONSTANT if the following values are not set for the supplied parameters for **init_sect_len** and **num_sections**:

	NS/2 format	**IBM EE format**	**NS/2 format (Windows 2000 and Windows NT only**	**IBM EE format (Windows 2000 and Windows NT only)**
init_sect_len	50	44	52	48
num_sections	16	9	16	9

The AP_DISPLAY_INVALID_CONSTANT is also returned when the following parameters are not set properly:

- **reserv2** must be set to NULL.
- **am_info** must be set to AP_NO.
- **tp_info** must be set to AP_NO.
- **adapter_info** must be set to AP_NO if NS/2 format is requested.
- **ms_info** must be set to AP_NO if NS/2 format is requested.

Host Integration Server and SNA Server Extensions

The Host Integration Server and SNA Server **DISPLAY** verb is compatible with the IBM ES for OS/2 version 1.0 **DISPLAY** verb. However, since IBM ES for OS/2 version 1.0 is a single-server system and Host Integration Server and SNA Server support multiple-server systems, the **DISPLAY** verb has been extended to allow the user to target a specific server running Host Integration Server or SNA Server by which the **DISPLAY** verb will be processed.

To direct a **DISPLAY** verb at a particular server running Host Integration Server or SNA Server, place the ASCII string CSEXTNID, followed by the computer name of the server running Host Integration Server or SNA Server at the start of the buffer pointed to by **buffer_ptr**. The computer name is a 32-byte ASCII string and can be zero or padded with spaces.

Because the local node identifier is configured on a per-node basis for IBM ES for OS/2 version 1.0 and can be different for each connection in Host Integration Server or SNA Server, Host Integration Server and SNA Server also allow you to specify an optional connection name. This is an 8-byte ASCII string, which is placed after the 32-byte computer name. Again, the string can be zero or padded with spaces. The following example illustrates the CSEXTNID extension:

```
csextnid computername 00000000000000000000 name
```

If you do not specify a connection name, Host Integration Server and SNA Server return information about the first connection configured for the Host Integration Server or SNA Server system.

If you do not specify a computer name, Host Integration Server and SNA Server will randomly choose a default **DISPLAY** computer and connection, unless a specific default **DISPLAY** connection has been configured on the server. These parameters can be configured with the Host Integration Server Manager or the Host Integration Server Administrator Client when using Host Integration Server 2000 or with the SNA Explorer or SNA Manager Client on SNA Server. **DISPLAY** will behave as if you specified the connection and the computer name of the server that owns the verb. For additional information about using default LUs, see the *Microsoft Host Integration Server 2000 online books*.

Host Integration Server and SNA Server also allow you to use **DISPLAY** to return a list of active servers. To do so, place the string CSEXTNIDCSLISTND in the **DISPLAY** buffer and set the supplied parameters **sna_global_info**, **lu62_info**, and so on, to AP_NO. The information is returned in the **DISPLAY** buffer in the following format:

```
#activenodes            - 2 bytes
  node_name 1           - 8 bytes
  box_name 1            - 32 bytes
  .
  node_name m
  box_name m
```

In the current version of Host Integration Server and SNA Server, **node_name** is always SNASERVR and **box_name** is the computer name of the server.

Differences by Information Type

Differences in the implementation of the **DISPLAY** verb between the Host Integration Server or SNA Server and IBM ES for OS/2 are described in this section by information type. For each information type, there is a topic that describes:

- The information defined by IBM ES for OS/2 version 1.0.
- The information returned by Host Integration Server 2000 and SNA Server 4.0.

> **Note** Host Integration Server and SNA Server do not support all of the information types supported by IBM ES for OS/2 version 1.0. If an information type is not listed in this section, it is not supported by Host Integration Server or SNA Server.

SNA Global Information

SNA global information is defined or returned as described here.

Defined by IBM ES for OS/2 Version 1.0

Information on SNA global information is provided in the **sna_global_info_sect** structure as defined below.

```
typedef struct sna_global_info_sect {
    unsigned char version;
    unsigned char release;
    unsigned char net_name[8];
    unsigned char pu_name[8];
    unsigned char node_id[4];
    type_product_set_id product_set_id;
    unsigned char alias_cp_name[8];
    unsigned char node_type;
    unsigned char cp_nau_addr;
    unsigned char corr_serv_disk;
    unsigned char reserved;
    unsigned char appc_version;
    unsigned char appc_release;
    unsigned char appc_fixlevel;
} SNA_GLOBAL_INFO_SECT;
```

Members

version
  Communications Manager Extended Edition version number.
release
  Communications Manager Extended Edition release number.
net_name
  Network name, first part of fully qualified control program (CP) name, in EBCDIC (type A).
pu_name
  PU name, second part of fully qualified CP name, in EBCDIC (type A).
node_id
  Four-byte hexadecimal exchange identifier.
product_set_id
  Computer product data.
alias_cp_name
  Node name (local name for CP) in ASCII.
node_type
  AP_NN, AP_EN, or AP_LEN.
cp_nau_addr
  CP NAU address where 0 means not used (an independent LU). Other legal values are 1 to 254.
corr_serv_disk
  Last four digits of corrective service disk number.
reserved
  Reserved field.
appc_version
  APPC version number.
appc_release
  APPC release number.
appc_fixlevel
  APPC fixlevel.

Returned by Host Integration Server and SNA Server

Information on SNA global information is provided in the **sna_global_info_sect** structure as defined below.

```
typedef struct sna_global_info_sect {
    unsigned char version;
    unsigned char release;
    unsigned char net_name[8];
```

```
    unsigned char pu_name[8];
    unsigned char node_id[4];
    type_product_set_id product_set_id;
    unsigned char alias_cp_name[8];
    unsigned char node_type;
    unsigned char cp_nau_addr;
    unsigned char corr_serv_disk;
    unsigned char reserved;
    unsigned char appc_version;
    unsigned char appc_release;
    unsigned char appc_fixlevel;
} SNA_GLOBAL_INFO_SECT;
```

Members

version
  Major operating system (OS) version number.
release
  Minor OS version number.
net_name
  Node network name in EBCDIC (type A).
pu_name
  PU name in EBCDIC (type A) associated with connection.
node_id
  Node identifier to send.
product_set_id
  Set to EBCDIC zeros.
alias_cp_name
  Node name, local name for the control program (CP), in ASCII.
node_type
  Set to AP_LEN.
cp_nau_addr
  CP NAU address where 0 means not used (an independent LU). Other legal values are 1 to 254.
corr_serv_disk
  Reserved field set to zero.
reserved
  Reserved field set to zero.
appc_version
  Host Integration Server or SNA Server major version number.
appc_release
  Host Integration Server or SNA Server minor version number.
appc_fixlevel
  Host Integration Server or SNA Server patchlevel.

Remarks

Host Integration Server and SNA Server return **version** and **release** as the major and minor OS version numbers from **GetVersion** (for Windows 2000, Windows NT, Windows 98, Windows 95, or Windows version 3.*x* systems) or **GetDosVersion** (for OS/2).

Because Host Integration Server 2000 and SNA Server have no information on the computer type, serial number, and manufacturer, **product_set_id** is set to EBCDIC zeros.

Host Integration Server and SNA Server do not support APPN node types, so the node type is returned as 1 (an AP_LEN node), and not 2 or 3 (AP_NN or AP_EN nodes), as defined by IBM ES for OS/2 version 1.0.

LU 6.2 Information

Information on LUs is provided in the **lu62_info_sect** structure as defined below.

```
typedef struct lu62_info_sect {
    unsigned long  lu62_init_sect_len;
    unsigned short num_lu62s;
    unsigned short total_lu62s;
} LU62_INFO_SECT;
```

Members

lu62_init_sect_len
Structure length.
num_lu62s
Number of configured LUs displayed.
total_lu62s
Total number of configured LUs.

For each configured LU, an **lu62_overlay** structure is provided as defined below.

```
typedef struct lu62_overlay {
    unsigned long  lu62_entry_len;
    unsigned long  lu62_overlay_len;
    unsigned char  lu_name[8];
    unsigned char  lu_alias[8];
    unsigned short num_plus;
    unsigned char  fqlu_name[17];
    unsigned char  default_lu;
    unsigned char  reserv3;
    unsigned char  lu_local_addr;
    unsigned short lu_sess_lim;
    unsigned char  max_tps;
    unsigned char  lu_type;
} LU62_OVERLAY;
```

Members

lu62_entry_len
Size of this LU entry.
lu62_overlay_len
This value contains **sizeof(**struct **lu62_overlay)**–**sizeof(lu62_entry_len)**.
lu_name
LU name (EBCDIC type A).
lu_alias
LU alias (ASCII).
num_plus
Number of partner LUs.
fqlu_name
Fully qualified LU name (EBCDIC type A).
default_lu
For local LU group, an LU equal to the **default_lu** is used if none is specified. Legal values are AP_NO and AP_YES.

On Host Integration Server and SNA Server, there is no concept of a default local LU. Therefore, the **default_lu** flag, which is set to AP_YES for the node in IBM ES for OS/2 version 1.0, is set to AP_NO for Host Integration Server and SNA Server.

lu_local_addr
NAU address, 0–254.
lu_sess_lim
Configured session limit, 0–255.
max_tps
Maximum number of TPs, 1–255.
lu_type

Always LU type 6.2.

For each configured LU, a **plu_62_overlay** structure for the partner LU is provided as defined below.

```
typedef struct plu62_overlay {
    unsigned long  plu62_entry_len;
    unsigned long  plu62_overlay_len;
    unsigned char  plu_alias[8];
    unsigned short num_modes;
    unsigned char  plu_un_name[8];
    unsigned char  fqplu_name[17];
    unsigned char  reserv3;
    unsigned char  plu_sess_lim;
    unsigned char  dlc_name[8];
    unsigned char  adapter_num;
    unsigned char  dest_addr_len;
    unsigned char  dest_addr[32];
    unsigned int   par_sess_supp:1;
    unsigned int   reserv4:7;
    unsigned int   def_already_ver:1;
    unsigned int   def_conv_sec:1;
    unsigned int   def_sess_sec:1;
    unsigned int   reserv5:5;
    unsigned int   act_already_ver:1;
    unsigned int   act_conv_sec:1;
    unsigned int   reserv6:6;
    unsigned int   implicit_part:1;
    unsigned int   reserv7:7;
} PLU62_OVERLAY;
```

Members

plu62_entry_len
Size of this partner LU entry.

plu62_overlay_len
This value contains **sizeof(**struct **plu62_overlay)**–**sizeof(plu62_entry_len)**.

plu_alias
Partner LU alias (ASCII).

num_modes
Number of modes.

plu_un_name
Partner LU uninterpreted name (EBCDIC).

fqplu_name
Fully qualified partner LU name (EBCDIC type A).

reserv3
Reserved field set to zero.

plu_sess_lim
Partner LU session limit, 0–255.

dlc_name
DLC name (ASCII).

adapter_num
DLC adapter number.

dest_addr_len
Length of destination adapter address.

dest_addr
Destination adapter address.

par_sess_supp
Bit 15 of a bitfield specifying parallel sessions. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

reserv4
Bits 8–14 of a bitfield specifying a reserved field set to zero.

def_already_ver
Bit 7 of a bitfield specifying whether the configured already verified option is supported. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

def_conv_sec

Bit 6 of a bitfield specifying whether the configured conversation security option is supported. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

def_sess_sec

Bit 5 of a bitfield specifying whether the configured session security option is supported. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

reserv5

Bits 0–4 of a bitfield specifying a reserved field set to zero.

act_already_ver

Bit 15 of a bitfield specifying whether the active already verified option is supported. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

act_conv_sec

Bit 14 of a bitfield specifying whether the active conversation security option is supported. Valid values are AP_NOT_SUPPORTED and AP_SUPPORTED.

reserv6

Bits 8–13 of a bitfield specifying a reserved field set to zero.

implicit_part

Bit 7 of a bitfield specifying whether this is an implicit partner. Valid values are AP_NO and AP_YES.

For partner LU group, **implicit_part** indicates the partner LU group was configured as an implicit primary logical unit (PLU).

reserv7

Bits 0–6 of a bitfield specifying a reserved field set to zero.

Remarks

Host Integration Server and SNA Server return information on all the configured LU 6.2s in the system, including the implicit PLU and all instances of implicit modes. IBM ES for OS/2 version 1.0 only returns information on those that are in use or have been in use.

For partner LU group, **implicit_part** indicates the partner LU group was configured as an implicit primary logical unit (PLU).

For mode group, **implicit_mode** bitfield returned in the mode_overlay structure indicates the mode group was configured as an implicit mode.

Session Information

Information on session information is provided in the **sess_info_sect** structure as defined below.

```
typedef struct sess_info_sect {
    unsigned long  sess_sect_len;
    unsigned short num_sessions;
    unsigned short total_sessions;
} SESS_INFO_SECT;
```

Members

sess_sect_len
The length of the initial session information section, including this parameter, up to the first session group. The length does not include any previous information sections.

num_sessions
The number of session groups returned by the DISPLAY verb into your program's buffer. This is the number of times the session group is repeated.

total_sessions
The total number of session groups. This number is the same as the number returned in the **num_sessions** member except when APPC has more information about session groups than it can place in the supplied buffer, in which case this number is larger.

For each session group, a **sess_overlay** structure for the session is provided as defined below.

```
typedef struct sess_overlay {
    unsigned long  sess_entry_len;
    unsigned long  reserv3;
    unsigned char  sess_id[8];
    unsigned long  conv_id[8];
    unsigned char  lu_alias[8];
    unsigned char  plu_alias[8];
    unsigned char  mode_name[8];
    unsigned short send_ru_size;
    unsigned short rcv_ru_size;
    unsigned short send_pacing_size;
    unsigned short rcv_pacing_size;
    unsigned char  link_id[12];
    unsigned char  daf;
    unsigned char  oaf;
    unsigned char  odai;
    unsigned char  sess_type;
    unsigned char  conn_type;
    unsigned char  reserv4;
    FPCID_OVERLAY  fpcid;
    unsigned char  cgid[4];
    unsigned char  fqlu_name[17];
    unsigned char  fqplu_name[17];
    unsigned char  pacing_type;
    unsigned char  reserv5;
 } SESS_OVERLAY;
```

Defined by IBM ES for OS/2 Version 1.0

Members

sess_entry_len
Size of this session group entry.

sess_id
The internal identifier of the session for which this information is displayed.

conv_id
The unique four-byte ID of the conversation currently using this session.

lu_alias
LU alias (ASCII).

plu_alias
Partner LU alias (ASCII).
mode_name
The name of the mode (EBCDIC).
send_ru_size
The maximum RU size used on this session and this **mode_name** for sending RUs.
rcv_ru_size
The maximum RU size used on this session and this **mode_name** for receiving RUs.
send_pacing_size
The size of the send pacing window on this session.
rcv_pacing_size
The size of the receive pacing window on this session.
link_id
Name of local logical link station.
daf
The destination address field for this session.
oaf
The origin address field for this session.
odai
The origin destination address indicator field for this session.
sess_type
The type of the session. The session type can be one of the following:

SSCP_PU_SESSION

This session is between a workstation physical unit and a host system services control point. This type of session exists if the local node contains a dependent LU, or if the session has been solicited in order to send alerts to the host.

SSCP_LU_SESSION

This session is between a dependent LU and a host system services control point.

LU_LU _SESSION

This session is between two LUs.

conn_type
Indicates whether the session activation protocol follows the rules for an independent LU or a dependent LU. The connection type can be one of the following:

AP_HOST_SESSION

For dependent LU protocols, the workstation LU is defined as dependent at the host, the host LU sends the session activation request (BIND), and each workstation LU can support only one session at a time.

AP_PEER_SESSION

For independent LU protocols, an LU can send a BIND, and can have multiple sessions to different partners, or parallel sessions to the same partner LU.

fq_pc_id
Fully qualified procedure correlation identifier of the session.
cgid
Unique identifier for the conversation group of the session.
fqlu_name
The fully-qualified LU name in EBCDIC (type A).
fqplu_name
The fully-qualified partner LU name in EBCDIC (type A).
pacing_type
The pacing type can be one of the following:

AP_FIXED

Fixed pacing.

AP_ADAPTIVE

Adaptive pacing.

Returned by Host Integration Server and SNA Server

Members

sess_entry_len
Size of this session group entry.
sess_id
The internal identifier of the session for which this information is displayed.
conv_id
The unique four-byte ID of the conversation currently using this session.
lu_alias
LU alias (ASCII).
plu_alias
Partner LU alias (ASCII).
mode_name
The name of the mode (EBCDIC).
mode_name
The name of the mode (EBCDIC).
send_ru_size
The maximum RU size used on this session and this **mode_name** for sending RUs.
rcv_ru_size
The maximum RU size used on this session and this **mode_name** for receiving RUs.
send_pacing_size
The size of the send pacing window on this session.
rcv_pacing_size
The size of the receive pacing window on this session.
link_id
Connection name.
daf
The destination address field for this session.
oaf
The origin address field for this session.
odai
The origin destination address indicator field for this session.
sess_type
The type of the session. The session type can be one of the following:

SSCP_PU_SESSION

This session is between a workstation physical unit and a host system services control point. This value is never returned by Host Integration Server or SNA Server.

SSCP_LU_SESSION

This session is between a dependent LU and a host system services control point.

LU_LU _SESSION

This session is between two LUs.

conn_type
Indicates whether the session activation protocol follows the rules for an independent LU or a dependent LU. The connection type can be one of the following:

AP_HOST_SESSION

For dependent LU protocols, the workstation LU is defined as dependent at the host, the host LU sends the session activation request (BIND), and each workstation LU can support only one session at a time.

AP_PEER_SESSION

For independent LU protocols, an LU can send a BIND, and can have multiple sessions to different partners, or parallel sessions to the same partner LU.

AP_BOTH_SESSION

fq_pc_id
Set to zero.

cgid

Set to zero.

type_of_pacing

The pacing type can be one of the following:

AP_FIXED

Fixed pacing.

AP_ADAPTIVE

Adaptive pacing. This value is never returned by Host Integration Server or SNA Server.

Active Link Information

Active link information is provided in the **link_info_sect** structure as defined below.

```
typedef struct link_info_sect {
    unsigned long  link_init_sect_len;
    unsigned short num_links;
    unsigned short total_links;
} LINK_INFO_SECT;
```

Members

link_init_sect_len
The length of the initial active link information section, including this parameter, up to the first link overlay group. The length does not include any previous information sections.

num_links
The number of active links returned by the DISPLAY verb into your program's buffer. This is the number of times the link overlay group is repeated.

total_sessions
The total number of active links. This number is the same as the number returned in the **num_links** member except when APPC has more information about active links than it can place in the supplied buffer, in which case this number is larger.

For each active link, a **link_overlay** structure for the active link is provided as defined below.

```
typedef struct link_overlay {
    unsigned long  link_entry_len;
    unsigned char  link_id[12];
    unsigned long  dlc_name[8];
    unsigned char  adapter_num;
    unsigned char  dest_addr_len;
    unsigned char  dest_addr[32];
    unsigned char  inbound_outbound;
    unsigned char  state;
    unsigned char  deact_link_flag;
    unsigned char  reserv3;
    unsigned short num_sessions;
    unsigned short ru_size;
    unsigned short reserv4;
    unsigned char  adj_fq_cp_name[17];
    unsigned char  adj_node_type;
    unsigned char  reserv5;
    unsigned char  cp_cp_sess_spt;
    unsigned char  conn_type;
    unsigned char  ls_role;
    unsigned char  line_type;
    unsigned char  tg_number;
    unsigned long  eff_capacity;
    unsigned char  conn_cost;
    unsigned char  byte_cost;
    unsigned char  propagation_delay;
    unsigned char  user_def_1;
    unsigned char  user_def_2;
    unsigned char  user_def_3;
    unsigned char  security;
    unsigned char  reserv6;
 } LINK_OVERLAY;
```

Defined by IBM ES for OS/2 Version 1.0

Members

link_entry_len
Size of this link entry.

link_id
Local logical link station name (EBCDIC).

dlc_name
Data link control (DLC) name set to one of the following:

ETHERAND

IBMTRNET

IBMPCNET

SDLC

TWINAX

X25DLC

adapter_num
Adapter number used by this link to connect to the adjacent node.

dest_addr_len
Length of the destination adapter address.

dest_addr
The destination adapter address.

inbound_outbound

state
The state of the link. The link state can be one of the following:

AP_CONALS_PND

The process to bring up the link has started but XID negotiation has not started.

AP_XID_PND

XID negotiation is in process.

AP_CONTACT_PND

XID negotiation has been completed but the final response from the DLC has not been received.

AP_CONTACTED

The link is fully functioning.

AP_DISC_PND

A request to disconnect the link has been issued to the DLC.

AP_DISC_RQ

The operator has requested that the link be disconnected.

deact_link_flag
Deactivate logical link.

num_sessions
Number of active sessions.

ru_size
RU size.

adj_fq_cp_name
Fully qualified **cp_name** in adjacent node.

adj_node_type
The adjacent node type (NN, EN, or LEN).

cp_cp_sess_spt
Specifies whether the link supports CP-CP sessions.

conn_type
Indicates whether the session activation protocol follows the rules for an independent LU or a dependent LU. The connection type can be one of the following:

AP_HOST_SESSION

For dependent LU protocols, the workstation LU is defined as dependent at the host, the host LU sends the session activation request (BIND), and each workstation LU can support only one session at a time.

AP_PEER_SESSION

For independent LU protocols, an LU can send a BIND, and can have multiple sessions to different partners, or parallel sessions to the same partner LU.

AP_BOTH_SESSION

ls_role
Specifies the link station role.
line_type
The line type.
tg_number
Transmission group number.
effective_capacity
Highest bit rate transmission effective capacity supported.
conn_cost
Relative cost per connection time using this link.
byte_cost
Relative cost of transmitting a byte over link.
propagation_delay
Indicates amount of time for signal to travel length of link. Set to one of the following:

AP_PROP_DELAY_MINIMUM

AP_PROP_DELAY_LAN

AP_PROP_DELAY_TELEPHONE

AP_PROP_DELAY_PKT_SWITCHED_NET

AP_PROP_DELAY_SATELLITE

AP_PROP_DELAY_MAXIMUM

user_def_1
User-defined TG characteristics.
user_def_2
User-defined TG characteristics.
user_def_3
User-defined TG characteristics.
security
The security value for this link. Set to one of the following:

AP_SEC_NONSECURE

AP_SEC_PUBLIC_SWITCHED_NETWORK

AP_SEC_UNDERGROUND_CABLE

AP_SEC_SECURE_CONDUIT

AP_SEC_GUARDED_CONDUIT

AP_SEC_ENCRYPTED

AP_SEC_GUARDED_RADIATION

Returned by Host Integration Server or SNA Server

Members

link_entry_len
Size of this link entry.
link_id
Connection name.
dlc_name
DLC name set to one of the following:

IBMTRNET

SDLC

DFT

X25DLC

adapter_num
Adapter number used by this link to connect to the adjacent node. Always set to zero.

dest_addr_len
Length of the destination adapter address.

dest_addr
The destination adapter address.

inbound_outbound

state
The state of the link. The link state can be one of the following:

AP_CONALS_PND

The process to bring up the link has started but XID negotiation has not started.

AP_XID_PND

XID negotiation is in process.

AP_CONTACT_PND

XID negotiation has been completed but the final response from the DLC has not been received.

AP_CONTACTED

The link is fully functioning.

AP_DISC_PND

A request to disconnect the link has been issued to the DLC.

AP_DISC_RQ

The operator has requested that the link be disconnected.

deact_link_flag
Deactivate logical link.

num_sessions
Number of active sessions.

ru_size
RU size.

adj_fq_cp_name
Fully qualified **cp_name** in adjacent node. Always set to EBCDIC spaces.

adj_node_type
The adjacent node type. Always set to AP_LEN.

cp_cp_sess_spt
Specifies whether the link supports CP-CP sessions. Always set to AP_NO.

conn_type
Indicates whether the session activation protocol follows the rules for an independent LU or a dependent LU. The connection type can be one of the following:

AP_HOST_SESSION

For dependent LU protocols, the workstation LU is defined as dependent at the host, the host LU sends the session activation request (BIND), and each workstation LU can support only one session at a time.

AP_PEER_SESSION

For independent LU protocols, an LU can send a BIND, and can have multiple sessions to different partners, or parallel sessions to the same partner LU.

ls_role
Specifies the link station role.

line_type
The line type.

tg_number
Transmission group number. Always set to zero.

effective_capacity
Highest bit rate transmission effective capacity supported. Always set to zero.

conn_cost
Relative cost per connection time using this link. Always set to zero.
byte_cost
Relative cost of transmitting a byte over link. Always set to zero.
propagation_delay
Indicates amount of time for signal to travel length of link. This parameter is always set to AP_PROP_DELAY_MAXIMUM.
user_def_1
User-defined TG characteristics. Always set to zero.
user_def_2
User-defined TG characteristics. Always set to zero.
user_def_3
User-defined TG characteristics. Always set to zero.
security
The security value for this link. Always set to AP_SEC_NONSECURE.

LU 0 to 3 Information

LU 0 to 3 information is provided in the **lu_0_3_info_sect** structure as defined below.

```
typedef struct lu_0_3_info_sect {
    unsigned long  lu_0_3_init_sect_len;
    unsigned short num_lu_0_3s;
} LU_0_3_ INFO_SECT;
```

Members

lu_0_3_init_sect_len
The length of the initial LU 0 to 3 information section, including this parameter, up to the first link overlay group. The length does not include any previous information sections.

num_links
The number of LU groups. This is the number of times the lu_0_3 overlay group is repeated.

For each configured LU, an **lu_0_3_overlay** structure for the LU is provided as defined below.

```
typedef struct lu_0_3_overlay {
    unsigned long  lu_0_3_entry_len;
    unsigned char  access_type;
    unsigned char  lu_type;
    unsigned char  lu_daf;
    unsigned char  lu_short_name;
    unsigned char  lu_long_name[8];
    unsigned char  sess_id[8];
    unsigned long  dlc_name[8];
    unsigned char  adapter_num;
    unsigned char  dest_addr_len;
    unsigned char  dest_addr[32];
    unsigned char  sscp_lu_sess_state;
    unsigned char  lu_lu_sess_state;
    unsigned char  link_id[12];
 } LU_0_3_OVERLAY;
```

Defined by IBM ES for OS/2 Version 1.0

Members

lu_3_3_entry_len
Size of this LU entry.

access_type
The access type (AP_3270 or AP_LUA).

lu_type
The LU type (AP_LU0, AP_LU1, AP_LU2, or AP_LU3).

lu_daf
The network addressable unit of the LU for which the information is displayed.

lu_short_name
The one-byte LU short name (ASCII).

lu_long_name
The eight-byte ASCII LU long name.

session_id
The LU-LU session ID.

dlc_name
DLC name set to one of the following:

ETHERAND

IBMTRNET

IBMPCNET

SDLC

TWINAX

X25DLC

adapter_number
The DLC adapter number for host link.
dest_addr_len
Length of the destination adapter address.
dest_addr
The destination adapter address.
sscp_lu_sess_state
Specifies the state of the SSCP-LU session.
lu_lu_sess_state
Specifies the state of the LU-LU session. The state can be one of the following:

AP_NOT_BOUND

The LU-LU session is not bound.

AP_BOUND

The LU-LU session is bound.

AP_BINDING

The LU-LU session is in the process of binding.

AP_UNBINDING

The LU-LU session is in the process of unbinding.

link_id
Name of local logical link station being used.

Returned by Host Integration Server and SNA Server

Members

lu_3_3_entry_len
Size of this LU entry.
access_type
The access type (AP_3270 or AP_LUA).
lu_type
The LU type (AP_LU0, AP_LU1, AP_LU2, or AP_LU3).
lu_daf
The network addressable unit of the LU for which the information is displayed.
lu_short_name
The one-byte ASCII LU short name.
lu_long_name
The eight-byte ASCII LU long name.
session_id
The LU-LU session ID.
dlc_name
DLC name set to one of the following:

IBMTRNET

SDLC

TWINAX

X25DLC

adapter_number
The DLC adapter number for host link. Always set to zero.
dest_addr_len
Length of the destination adapter address.
dest_addr
The destination adapter address.
sscp_lu_sess_state

Specifies the state of the SSCP-LU session.

lu_lu_sess_state

Specifies the state of the LU-LU session. The state can be one of the following:

AP_NOT_BOUND

The LU-LU session is not bound.

AP_BOUND

The LU-LU session is bound.

AP_BINDING

The LU-LU session is in the process of binding.

AP_UNBINDING

The LU-LU session is in the process of unbinding.

link_id

Name of connection.

System Default Information

System default information is defined or returned as described here.

Defined by IBM ES for OS/2 Version 1.0

Members

default_mode_name
Mode name used for undefined mode name is sent or received.
default_local_lu_name
Alias or local default LU.
implicit_partner_lu_support
Indicates if implicit partner LU support is enabled.
maximum_held_alerts
Number of alerts that will be held by NS/2 if there is no active link to a focal point.
default_tp_conversation_security_rqd
Specifies if conversation security is used for default TPs.
maximum_mc_ll_send_size
Maximum length of a logical record used on a mapped conversation for sending data to either the inbound or outbound implicit remote LU.
directory_for_inbound_attaches
Name of OS/2 directory used by Attach Manager.
default_tp_operation
Set to one of the following:

QUEUED_OPERATOR_STARTED

QUEUED_OPERATOR_PRELOADED

QUEUED_AM_STARTED

NONQUEUED_AM_STARTED

default_tp_program_type
Set to one of the following:

BACKGROUND

FULL_SCREEN

PRESENTATION_MANAGER

VIO_WINDOWABLE

Returned by Host Integration Server and SNA Server

Members

default_mode_name
Always set to NULL.
default_local_lu_name
Always set to spaces.
implicit_partner_lu_support
Always set to NO.
maximum_held_alerts
Always set to zero.
default_tp_conversation_security_rqd
Always set to NO.
maximum_mc_ll_send_size
Always set to 16384.
directory_for_inbound_attaches
Always returned * and indicates that the current path should be used.
default_tp_operation
Always set to QUEUED_AM_STARTED.
default_tp_program_type
Always set to FULL_SCREEN.

LU 6.2 Definition Information

There are no differences for this information type.

Partner Definition Information

Partner definition information is defined or returned as described here.

Defined by IBM ES for OS/2 Version 1.0

Members

maximum_mc_ll_send_size
Maximum length of a logical record used on a mapped conversation for sending data to the partner LU.
number_of_alternate_aliases
Specifies number of alternate aliases configured.

Returned by Host Integration Server and SNA Server

maximum_mc_ll_send_size
Always set to 16384.
number_of_alternate_aliases
Always set to zero.

Mode Definition Information

Mode definition information is defined or returned as described here.

Defined by IBM ES for OS/2 Version 1.0

Members

cos_name
Name of class of service.

Returned by Host Integration Server and SNA Server

Members

cos_name
Set to EBCDIC spaces.

Link Definition Information

Link definition information is provided in the **link_def_info_sect** structure as defined below.

```
typedef struct link_def_info_sect {
    unsigned long  link_def_init_sect_len;
    unsigned short num_link_def;
    unsigned short total_link_def;
} LINK_DEF_INFO_SECT;
```

Members

link_def_init_sect_len
The length of the initial link definition information section, including this parameter, up to the first link definition overlay group. The length does not include any previous information sections.

num_link_def
The number of link definitions returned by the DISPLAY verb into your program's buffer. This is the number of times the link definition overlay is repeated.

total_sessions
The total number of link definitions. This number is the same as the number returned in the **num_link_def** member except when APPC has more information about link definitions than it can place in the supplied buffer, in which case this number is larger.

For each link definition, a **link_def_overlay** structure for the link definition is provided as defined below.

```
typedef struct link_def_overlay {
    unsigned long  link_def_entry_len;
    unsigned char  link_name[8];
    unsigned char  adj_fq_cp_name[17];
    unsigned char  adj_node_type;
    unsigned long  dlc_name[8];
    unsigned char  adapter_num;
    unsigned char  dest_addr_len;
    unsigned char  dest_addr[32];
    unsigned char  preferred_nn_server;
    unsigned char  auto_act_link;
    unsigned char  tg_number;
    unsigned char  lim_res;
    unsigned char  solicit_sscp_session;
    unsigned char  initself;
    unsigned char  bind_support;
    unsigned char  ls_role;
    unsigned char  line_type;
    unsigned long  eff_capacity;
    unsigned char  conn_cost;
    unsigned char  byte_cost;
    unsigned char  propagation_delay;
    unsigned char  user_def_1;
    unsigned char  user_def_2;
    unsigned char  user_def_3;
    unsigned char  security;
    unsigned char  reserv;
 } LINK_OVERLAY;
```

Defined by IBM ES for OS/2 Version 1.0

Members

link_def_entry_len
Size of this link definition entry.

link_name
Local logical link station name (EBCDIC).

dlc_name
Data link control (DLC) name set to one of the following:

ETHERAND

IBMTRNET

IBMPCNET

SDLC

TWINAX

X25DLC

adj_fq_cp_name
Fully qualified **cp_name** in adjacent node.
adj_node_type
The adjacent node type (AP_ADJACENT_NN, AP_LEARN, or AP_LEN).
adapter_num
DLC adapter number used by this link.
dest_addr_len
Length of the destination adapter address.
dest_addr
The destination adapter address.
cp_cp_sess_spt
Specifies whether the link supports CP-CP sessions.
preferred_nn_server
Indicates if this is the preferred NN server.
auto_act_link
Indicates if the link should be automatically activated.
tg_number
Transmission group number.
lim_res
Indicates if this is a limited resource.
solicit_sscp_session
Indicates whether to solicit an SSCP session.
initself
Indicates if the node supports receiving INIT_SELF over this link.
bind_support
Indicates whether BIND support is available.
ls_role
Specifies the link station role.
line_type
The line type.
effective_capacity
Highest bit rate transmission effective capacity supported.
conn_cost
Relative cost per connection time using this link.
byte_cost
Relative cost of transmitting a byte over link.
propagation_delay
Indicates amount of time for signal to travel length of link. Set to one of the following:

AP_PROP_DELAY_MINIMUM

AP_PROP_DELAY_LAN

AP_PROP_DELAY_TELEPHONE

AP_PROP_DELAY_PKT_SWITCHED_NET

AP_PROP_DELAY_SATELLITE

AP_PROP_DELAY_MAXIMUM

user_def_1
User-defined TG characteristics.
user_def_2
User-defined TG characteristics.

user_def_3
User-defined TG characteristics.
security
The security value for this link. Set to one of the following:

AP_SEC_NONSECURE

AP_SEC_PUBLIC_SWITCHED_NETWORK

AP_SEC_UNDERGROUND_CABLE

AP_SEC_SECURE_CONDUIT

AP_SEC_GUARDED_CONDUIT

AP_SEC_ENCRYPTED

AP_SEC_GUARDED_RADIATION

Returned by Host Integration Server and SNA Server

Members

link_def_entry_len
Size of this link definition entry.
link_name
Local logical link station name (EBCDIC).
dlc_name
Data link control (DLC) name set to one of the following:

IBMTRNET

SDLC

DFT

X25DLC

adj_fq_cp_name
Fully qualified **cp_name** in adjacent node. Always set to EBCDIC spaces.
adj_node_type
The adjacent node type. Always set to AP_LEN.
adapter_num
DLC adapter number used by this link. Always set to zero.
dest_addr_len
Length of the destination adapter address.
dest_addr
The destination adapter address.
cp_cp_sess_spt
Specifies whether the link supports CP-CP sessions. Always set to AP_NO.
preferred_nn_server
Indicates if this is the preferred NN server.
auto_act_link
Indicates if the link should be automatically activated.
tg_number
Transmission group number. Always set to zero.
lim_res
Indicates if this is a limited resource.
solicit_sscp_session
Indicates whether to solicit an SSCP session.
initself
Indicates if the node supports receiving INIT_SELF over this link.
bind_support
Indicates whether BIND support is available.
ls_role
Specifies the link station role.
line_type
The line type.

effective_capacity
Highest bit rate transmission effective capacity supported. Always set to zero.
conn_cost
Relative cost per connection time using this link. Always set to zero.
byte_cost
Relative cost of transmitting a byte over link. Always set to zero.
propagation_delay

Indicates amount of time for signal to travel length of link. Set to one of the following: Always set to AP_PROP_DELAY_MAXIMUM.

user_def_1
User-defined TG characteristics. Always set to zero.
user_def_2
User-defined TG characteristics. Always set to zero.
user_def_3
User-defined TG characteristics. Always set to zero.
security
The security value for this link. Always set to AP_SEC_NONSECURE.

Management Services Information

Information on management services is provided in the **ms_info_sect** structure as defined below.

```
typedef struct ms_info_sect {
    unsigned long  ms_init_sect_len;
    unsigned char  held_mds_mu_alerts;
    unsigned char  held_nmvt_alerts;
    unsigned short num_fps;
    unsigned short total_fps;
    unsigned short num_ms_appls;
    unsigned short total_ms_appls;
    unsigned short num_act_trans;
    unsigned short total_act_trans;
} MS_INFO_SECT;
```

Members

ms_init_sect_len
The length of the initial MS information section, including this parameter, up to the first MS focal point group. The length does not include any previous information sections.

held_mds_mu_alerts
The number of management service MDS alerts being held that will be sent to the management service alert focal point (FP) when one becomes available.

held_nmvt_alerts
The number of management service NMVT alerts being held that will be sent to the management service alert focal point (FP) when one becomes available.

num_fps
The number of management service focal points (MS FPs) for which the information listed under MS Focal Point Group is returned. This is the number of times the information group is repeated.

total_fps
The total number of management service focal points for which APPC has information. This number is the same as the number returned in the **num_fps** member except when APPC has more information about management service focal points than it can place in the supplied buffer, in which case this number is larger.

num_ms_appls
The number of registered MS applications for which the information listed under Registered MS Application Group is returned. This is the number of times the information group is repeated.

total_ms_appls
The total number of registered MS applications for which APPC has information. This number is the same as the number returned in the **num_ms_appls** member except when APPC has more information about registered MS applications than it can place in the supplied buffer, in which case this number is larger.

num_act_trans
The number of MS active transactions for which the information listed under MS Active Transaction Group is returned. This is the number of times the information group is repeated.

total_act_trans
The number or MS active transactions for which APPC has information. This number is the same as the number returned in the **num_act_trans** member except when APPC has more information about registered MS active transactions than it can place in the supplied buffer, in which case this number is larger.

For each local and remore management service focal point group, an **ms_fp_overlay** structure for the focal point group is provided as defined below.

```
typedef struct ms_fp_overlay {
    unsigned long  ms_fp_entry_len;
    unsigned char  ms_appl_name[8];
    unsigned char  ms_category[4];
    unsigned char  fp_fq_cp_name[17];
    unsigned char  bkup_appl_name[8];
    unsigned char  bkup_fp_fq_cp_name[17];
    unsigned char  reserv1;
    unsigned char  fp_type;
    unsigned char  fp_status;
    unsigned char  fp_routing;
```

```
} MS_FP_OVERLAY;
```

Members

ms_fp_entry_len
Size of this management service focal point information entry.

ms_appl_name
The management service application name of the current active focal point (EBCDIC).

ms_category
The management service category.

fp_fq_cp_name
The fully qualified control point name of the node on which the current (active) management service focal point is located (EBCDIC). If the local node has no focal point, a value of all EBCDIC space characters (0x40) is returned.

bkup_appl_name
The management service application name of the backup focal point, if one is known (EBCDIC).

bkup_fp_fq_cp_name
The fully qualified control point name of the node on which the backup management service focal point is located, if one is known (EBCDIC). If the local node has no backup focal point, a value of all EBCDIC space characters (0x40) is returned.

fp_type
The type of the focal point for the local management service entry point node. The focal point type depends on how the focal point-end point relationship was established, and on whether the local node is configured as an NN, EN, or LEN node (an EN without CP-CP session support). The type can be one of the following:

AP_EXPLICIT_PRIMARY_FP

The current focal point type is explicit primary.

AP_BACKUP_FP

The current focal point type is backup.

AP_DEFAULT_PRIMARY_FP

The current focal point type is default primary.

AP_DOMAIN_FP

The current focal point type is domain.

AP_HOST_FP

The current focal point type is host.

AP_NO_FP

Currently the local node has no focal point.

fp_status
The status of the management service focal point. The status can be one of the following:

AP_NOT_ACTIVE

The focal point has been acquired, but has since become unavailable.

AP_ACTIVE

The remote focal point has been acquired and is available.

AP_PENDING

A request has been sent to a remote primary or backup focal point to acquire that FP, and its reply has not yet been received.

AP_NEVER_ACTIVE

The focal point has never been acquired, but one or more registered management service applications have requested focal point information.

fp_routing
The routing used to send unsolicited requests to the management service focal point when the local node is an EN. Note that requests from an NN are always sent directly to the focal point.

The routing can be one of the following:

AP_DEFAULT

Unsolicited management service requests destined for the focal point are sent from the EN to its serving NN for forwarding to the focal point.

AP_DIRECT

Unsolicited management service requests destined for the focal point are sent directly to the focal point.

Remarks

When a program registers an management service application name, it can request focal point information. When APPC acquires the focal point, it passes the program the focal point information, which includes the type of routing to use to send unsolicited management service requests to the focal point.

APPC TP Verbs

This section describes the APPC TP verbs:

GET_TP_PROPERTIES

SET_TP_PROPERTIES

TP_ENDED

TP_STARTED

The description of each verb provides:

- A definition of the verb.
- The structure defining the VCB used by the verb. The structure is contained in the WINAPPC.H file. The length of each VCB field is in bytes. Fields beginning with **reserv** (for example, **reserv2**) are reserved.
- The parameters (VCB fields) supplied to and returned by APPC. A description of each parameter is provided, along with its possible values and other information.
- The conversation state(s) in which the verb can be issued.
- The state(s) to which the conversation can change upon return from the verb. Conditions that do not cause a state change are not noted. For example, parameter checks and state checks do not cause a state change.
- Additional information describing the verb.

Most parameters supplied to and returned by APPC are hexadecimal values. To simplify coding, these values are represented by meaningful symbolic constants, which are established by **#define** statements in the WINAPPC.H header file. For example, the **opcode** (operation code) member of the **mc_send_data** structure used by the **MC_SEND_DATA** verb is the hexadecimal value represented by the symbolic constant AP_M_SEND_DATA. Use only the symbolic constants when writing TPs.

GET_TP_PROPERTIES

The **GET_TP_PROPERTIES** verb returns attributes of the TP and the current transaction.

The following structure describes the verb control block used by the **GET_TP_PROPERTIES** verb.

```
struct get_tp_properties {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned char   tp_name[64];
    unsigned char   lu_alias[8];
    unsigned char   luw_id[26];
    unsigned char   fqlu_name[17];
    unsigned char   reserve3[10];
    unsigned char   user_id[10];
    unsigned char   prot_luw_id[26];
    unsigned char   pwd[10];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_GET_TP_PROPERTIES.

opext
Supplied parameter. Specifies the verb operation extension. If the AP_EXTD_VCB bit is set, this indicates that the **get_tp_properties** structure includes the **prot_luw_id** member used for Sync Point support. Otherwise the verb control block ends immediately after the **user_id** member.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

tp_name
Returned parameter. Supplies the TP name of the TP that issued the verb. The name is returned as a 64-byte EBCDIC string, padded on the right with EBCDIC spaces.

lu_alias
Returned parameter. Supplies the alias name assigned to the local LU. It is returned as an 8-byte ASCII string padded on the right with ASCII spaces.

luw_id
Returned parameter. Supplies the unprotected logical unit-of-work identifier for the transaction in which the TP is participating. Several TPs can be involved in a transaction. This identifier, which is assigned on behalf of the TP initiating the transaction, allows the conversation that makes up the transaction to be logically connected.

The **luw_id** can be represented as an **luw_id_overlay** structure with the following fields:

```
typedef struct   luw_id_overlay {
    unsigned char  fqla_name_len;
    unsigned char  fqla_name[17];
    nsigned char   instance[6];
    unsigned char  sequence[2];
} LUW_ID_OVERLAY;
```

luw_id.fqla_name_len

A 1-byte length of the fully qualified LU name for the LU of the originating TP.

luw_id.fqla_name

The fully qualified name of the LU for the originating TP. The name is returned as a 17-byte EBCDIC string, consisting of the NETID, a period, and the LU name. If the length of the name is fewer than 17 bytes, the **instance** and **sequence** numbers follow immediately. (Note that because of this, you should not use the fields of the **luw_id_overlay** structure to access those values. These are provided for compatibility only).

luw_id.instance

A 6-byte string uniquely generated by the LU for the originating TP.

luw_id.sequence

A 2-byte number that indicates the segment of unit-of-work. (This is always set to 1, if Sync Point is not supported.)

If the **luw_id** length is fewer than 26 bytes, it is padded on the right with EBCDIC spaces.

fqlu_name
Returned parameter. Supplies the fully qualified name of the local LU. The name is returned as a 17-byte EBCDIC string, consisting of the NETID, a period, and the LU name. The name is padded on the right with EBCDIC spaces.

reserve3

A reserved field.

user_id
Supplied parameter. Indicates the **user_id** supplied by the initiating TP in the allocation request. The name is supplied as a 10-byte EBCDIC string, padded on the right with EBCDIC spaces.

prot_luw_id
Returned parameter. Contains the protected logical unit-of-work identifier for the transaction in which the TP is participating, if the conversation was allocated with **synclevel** Sync Point.

Several TPs can be involved in a transaction. This identifier, which is assigned on behalf of the TP initiating the transaction, allows the conversation that makes up the transaction to be logically connected.

The **prot_luw_id** can be represented as an **luw_id_overlay** structure with the following fields:

```
typedef struct   luw_id_overlay {
    unsigned char  fqla_name_len;
    unsigned char  fqla_name[17];
    nsigned char   instance[6];
    unsigned char  sequence[2];
} LUW_ID_OVERLAY;
```

luw_id.fqla_name_len

A 1-byte length of the fully qualified LU name for the LU of the originating TP.

luw_id.fqla_name

The fully qualified name of the LU for the originating TP. The name is returned as a 17-byte EBCDIC string, consisting of the NETID, a period, and the LU name. If the length of the name is fewer than 17 bytes, the **instance** and **sequence** numbers follow immediately. (Note that because of this, you should not use the fields of the **luw_id_overlay** structure to access those values. These are provided for compatibility only).

luw_id.instance

A 6-byte string uniquely generated by the LU for the originating TP.

luw_id.sequence

A 2-byte number that indicates the segment of unit-of-work. (This is always set to 1, if Sync Point is not supported.)

If the **prot_luw_id** length is fewer than 26 bytes, it is padded on the right with EBCDIC spaces.

pwd
Supplied parameter. Contains the password of the **user_id** of the initiating TP in the allocation request. The password is supplied as a 10-byte EBCDIC string, padded on the right with EBCDIC spaces.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE or MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** or **MC_ALLOCATE** request.

When **ALLOCATE** or **MC_ALLOCATE** produces this return code for a system configured with multiple nodes using Microsoft® Host Integration Server 2000 or SNA Server 4.0, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_TP_BUSY
Primary return code; the local TP has issued a call to APPC while APPC was processing another call for the same TP. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **tp_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

This verb relates to the TP rather than a specification conversation, so the TP can issue the verb in any state. There is no state change.

The **luw_id** member contains fields for **fqla_name_len** (the length of the fully qualified LU name of the LU originating the TP), **fqla_name** (the fully qualified name of the LU originating the TP), **instance** (generated uniquely by the LU originating the TP), and **sequence** (always set to 1 and indicating the segment of unit-of-work).

SET_TP_PROPERTIES

The **SET_TP_PROPERTIES** verb allows a TP to set its logical unit-of-work identifiers (LUWIDs) to either an existing value, by providing the LUWIDs, or request that the SNA server generate new ones and use them from then on. When the LUWID is generated by the SNA server, it is guaranteed to be unique. This verb is used only if Sync Point support is enabled.

The following structure describes the verb control block used by the **SET_TP_PROPERTIES** verb.

```
struct set_tp_properties {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned char   set_prot_id;
    unsigned char   new_prot_id;
    unsigned char   prot_id[26];
    unsigned char   set_unprot_id;
    unsigned char   new_unprot_id;
    unsigned char   unprot_id[26];
    unsigned char   set_user_id;
    unsigned char   reserv3;
    unsigned char   user_id[10];
    unsigned char   reserv4[10];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_SET_TP_PROPERTIES.

opext
Supplied parameter. Specifies the verb operation extension. The AP_EXTD_VCB bit must be set to indicate that the **set_tp_properties** structure requires Sync Point support.

reserv2

A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

set_prot_id
Supplied parameter. Indicates whether the **prot_id** member should be modified. Legal values are AP_YES or AP_NO.

new_prot_id
Supplied parameter. Indicates whether Host Integration Server or SNA Server should use the supplied **prot_id** LUWID member or create a new LUWID. Legal values are AP_YES (create a new LUWID) or AP_NO (use the supplied LUWID).

prot_id
This member is the protected logical unit-of-work identifier for the transaction in which the TP is participating. It is ignored if **set_prot_id** is AP_NO. It is a supplied parameter if **new_unprot_id** is AP_NO or a returned parameter if **new_unprot_id** is AP_YES.

Several TPs can be involved in a transaction. This identifier, which is assigned on behalf of the TP initiating the transaction, allows the conversation that makes up the transaction to be logically connected.

The **prot_id** can be represented as an **luw_id_overlay** structure with the following fields:

```
typedef struct   luw_id_overlay {
    unsigned char  fqla_name_len;
```

```
    unsigned char  fqla_name[17];
    nsigned char   instance[6];
    unsigned char  sequence[2];
} LUW_ID_OVERLAY;
```

luw_id.fqla_name_len

A 1-byte length of the fully qualified LU name for the LU of the originating TP.

luw_id.fqla_name

The fully qualified name of the LU for the originating TP. The name is returned as a 17-byte EBCDIC string, consisting of the NETID, a period, and the LU name. If the length of the name is fewer than 17 bytes, the **instance** and **sequence** numbers follow immediately. (Note that because of this, you should not use the fields of the **luw_id_overlay** structure to access those values. These are provided for compatibility only).

luw_id.instance

A 6-byte string uniquely generated by the LU for the originating TP.

luw_id.sequence

A 2-byte number that indicates the segment of unit-of-work. (This is always set to 1 if Sync Point is not supported.)

If the **luw_id** length is fewer than 26 bytes, it is padded on the right with EBCDIC spaces.

set_unprot_id
Supplied parameter. Indicates whether the **unprot_id** member should be modified. Legal values are AP_YES or AP_NO.

new_unprot_id
Supplied parameter. Indicates whether Host Integration Server or SNA Server should use the supplied **unprot_id** LUWID member or create a new LUWID. Legal values are AP_YES (create a new LUWID) or AP_NO (use the supplied LUWID).

unprot_id
This member is the unprotected logical unit-of-work identifier for the transaction in which the TP is participating. It is ignored if **set_unprot_id** is AP_NO. It is a supplied parameter if **new_unprot_id** is AP_NO or a returned parameter if **new_unprot_id** is AP_YES.

Several TPs can be involved in a transaction. This identifier, which is assigned on behalf of the TP initiating the transaction, allows the conversation that makes up the transaction to be logically connected.

The **prot_id** can be represented as an **luw_id_overlay** structure with the following fields:

```
typedef struct   luw_id_overlay {
    unsigned char  fqla_name_len;
    unsigned char  fqla_name[17];
    nsigned char   instance[6];
    unsigned char  sequence[2];
} LUW_ID_OVERLAY;
```

luw_id.fqla_name_len

A 1-byte length of the fully qualified LU name for the LU of the originating TP.

luw_id.fqla_name

The fully qualified name of the LU for the originating TP. The name is returned as a 17-byte EBCDIC string, consisting of the NETID, a period, and the LU name. If the length of the name is fewer than 17 bytes, the **instance** and **sequence** numbers follow immediately. (Note that because of this, you should not use the fields of the **luw_id_overlay** structure to access those values. These are provided for compatibility only).

luw_id.instance

A 6-byte string uniquely generated by the LU for the originating TP.

luw_id.sequence

A 2-byte number that indicates the segment of unit-of-work. (This is always set to 1 if Sync Point is not supported.)

If the **luw_id** length is fewer than 26 bytes, it is padded on the right with EBCDIC spaces.

set_user_id

Supplied parameter. Indicates whether the **user_id** member should be modified. Legal values are AP_YES or AP_NO.

reserve3

A reserved field.

user_id

Supplied parameter. Indicates the **user_id** that should be used by the initiating TP in the allocation request. The name is a 10-byte EBCDIC string, padded on the right with EBCDIC spaces. This parameter is ignored if **set_user_id** is AP_NO.

reserve4

A reserved field.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE or MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** or **MC_ALLOCATE** request.

When **ALLOCATE** or **MC_ALLOCATE** produces this return code for an Host Integration Server or SNA Server system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_TP_BUSY

Primary return code; the local TP has issued a call to APPC while APPC was processing another call for the same TP. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **tp_id**.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

This verb relates to the TP rather than a specification conversation, so the TP can issue the verb in any state. There is no state change.

The **prot_id** and **unprot_id** members contain fields for **fqla_name_len** (the length of the fully qualified LU name of the LU originating the TP), **fqla_name** (the fully qualified name of the LU originating the TP), **instance** (generated uniquely by the LU originating the TP), and **sequence** (always set to 1 and indicating the segment of unit-of-work).

It is the responsibility of the application (the Sync Point support component) to transmit the new LUWID PS header to the partner Sync Point support when the protected LUWID is changed. Similarly, when the new LUWID PS header is received, the application must inform the LU by issuing a **SET_TP_PROPERTIES** verb.

TP_ENDED

The **TP_ENDED** verb is issued by both the invoking and invoked TP, and notifies APPC that the TP is ending.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use the WinAsyncAPPC function rather than the blocking version of this call.

The following structure describes the verb control block used by the **TP_ENDED** verb.

```
struct tp_ended {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned char   type;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_TP_ENDED.

opext
Supplied parameter. Specifies the verb operation extension. This field is not used by the **TP_ENDED** verb.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

type
Supplied parameter. Specifies the type of termination to be performed. The following are allowed values:

- AP_HARD indicates that all active verbs for the TP are terminated; the session(s) being used by the conversation(s) are ended. Both the local TP and the partner TP can receive conversation failure return codes (AP_DEALLOC_ABEND for mapped conversations and AP_DEALLOC_ABEND_PROG for basic conversations).
- AP_SOFT indicates that the TP waits for all active verbs to complete; the session being used by the conversation remains active.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_TP_ID

Secondary return code; APPC did not recognize the **tp_id** as an assigned TP identifier.

AP_BAD_TYPE

Secondary return code; the specified **type** value was not recognized by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).

- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_TP_BUSY
Primary return code; the local TP has issued a call to APPC while APPC was processing another call for the same TP. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **tp_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

In response to **TP_ENDED**, APPC frees the resources used by the TP. After this verb executes, the TP identifier is no longer valid; the TP cannot issue any more APPC conversation verbs.

The conversation can be in any state when the TP issues this verb.

If the conversation is in SEND state, **TP_ENDED** performs the function of DEALLOCATE or MC_DEALLOCATE with **dealloc_type** set to AP_FLUSH.

If the conversation is in a state other than RESET or SEND, **TP_ENDED** performs the function of **DEALLOCATE** or **MC_DEALLOCATE** with **dealloc_type** set to AP_ABEND (for a mapped conversation) or AP_ABEND_PROG (for a basic conversation).

After successful execution (**primary_rc** is AP_OK), there is no APPC state.

TP_STARTED

The **TP_STARTED** verb is issued by the invoking TP, and notifies APPC that the TP is starting.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use the WinAsyncAPPC function rather than the blocking version of this call.

The following structure describes the verb control block used by the **TP_STARTED** verb.

```
struct tp_started {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   lu_alias[8];
    unsigned char   tp_id[8];
    unsigned char   tp_name[64];
    unsigned char   syncpoint_rqd;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_TP_STARTED.

opext
Supplied parameter. Specifies the verb operation extension. If the AP_EXTD_VCB bit is set, this indicates that the **tp_started** structure includes the **syncpoint_rqd** member used for Sync Point support. Otherwise, the verb control block ends immediately after the **tp_name** member.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

lu_alias
Supplied parameter. Specifies the alias by which the local LU is known to the local TP.

The name must match an LU alias established during configuration. APPC checks the LU alias against the current Host Integration Server or SNA Server configuration file. Due to the client/server architecture used by Host Integration Server and SNA Server, however, this parameter is not validated until an ALLOCATE or MC_ALLOCATE is performed.

This parameter is an 8-byte ASCII character string. It can consist of the following ASCII characters:

- Uppercase letters
- Numerals from 0 through 9
- Spaces
- Special characters $, #, % and @

The first character of this string cannot be a space.

If the value of this parameter is fewer than eight bytes in length, pad it on the right with ASCII spaces (0x20).

To use an LU from the default LU pool, set this field to eight hexadecimal zeros. For more information, see Default LUs.

tp_id
Returned parameter. Identifies the newly-established TP.

tp_name
Supplied parameter. Specifies the name of the local TP.

Under the Host Integration Server and SNA Server implementation of APPC, this parameter is ignored when issued by **TP_STARTED**. However, this parameter is required if the program runs under the IBM ES for OS/2 version 1.0 implementation of APPC.

This parameter is a 64-byte EBCDIC character string and is case-sensitive. The **tp_name** parameter can consist of the following EDCDIC characters:

- Uppercase and lowercase letters
- Numerals from 0 through 9
- Special characters $, #, @, and period (.)

If the TP name is fewer than 64 bytes in length, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention for a service TP name is up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F.

syncpoint_rqd
This optional parameter is only applicable if the AP_EXTD_VCB bit is set in the **opext** parameter and Sync Point services are required.

- AP_YES if Sync Point is required.
- AP_NO if Sync Point is not required.

Return Codes

AP_OK
Primary return code; the verb executed successfully.
AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE or MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with **TP_STARTED** is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** or **MC_ALLOCATE** request.

When **ALLOCATE** or **MC_ALLOCATE** produces this return code for a system configured with multiple nodes using Host Integration Server or SNA Server, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_TP_BUSY
Primary return code; the local TP has issued a call to APPC while APPC was processing another call for the same TP.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The

operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

In response to **TP_STARTED**, APPC generates a TP identifier for the invoking TP. This identifier is a required parameter for subsequent APPC verbs issued by the invoking TP.

This must be the first APPC verb issued by the invoking TP. Consequently, no prior APPC state exists.

If the verb executes successfully (**primary_rc** is AP_OK), the state changes to RESET.

Default LUs

Any LU can be configured to be in a pool of default local LUs available for use by invoking TPs.

For a user or group who will be using TPs, 5250 emulators, and/or APPC applications, you can assign a default local APPC LU and a default remote APPC LU. If the invoking TP specifies the LU alias that it uses (in TP_STARTED), that LU alias must match a local APPC LU alias on the supporting SNA server. If the invoking TP leaves the LU alias blank in **TP_STARTED**, one of two methods for designating a default LU must be carried out on the supporting SNA server:

- Assign a default local APPC LU to the user or group that starts the invoking TP (that is, the user or group logged on at the system from which **TP_STARTED** is issued).

  —or—

- Designate one or more LUs as members of the default outgoing local APPC LU pool. The SNA server first attempts to determine the default local APPC LU of the associated user or group, then attempts to assign an available LU from the default outgoing local APPC LU pool; if these attempts fail, the SNA server rejects the request.

In AS/400 environments, the ability to assign default APPC LUs to users or groups is especially useful because it gives the administrator centralized control over these LU assignments. In such environments, for each user or group, assign both a default local APPC LU and a default remote APPC LU. Assigning a default local APPC LU for each user fulfills the normal AS/400 procedure of assigning local LUs on a per-user basis. Assigning a default remote APPC LU is equivalent to assigning a default AS/400 for the user to connect to, since the remote LU designates the AS/400. By making these assignments, the administrator can centrally control the default AS/400 that a 5250 emulator user connects to.

For more information, see the *Microsoft Host Integration Server 2000 online books*.

APPC Conversation Verbs

This section describes the APPC conversation verbs. The description of each verb provides:

- A definition of the verb.
- The structure defining the VCB used by the verb. The structure is contained in the WINAPPC.H file. The length of each VCB field is in bytes. Fields beginning with reserv (for example, reserv2) are reserved.
- The parameters (VCB fields) supplied to and returned by APPC. A description of each parameter is provided, along with its possible values and other information.
- The conversation state(s) in which the verb can be issued.
- The state(s) to which the conversation can change upon return from the verb. Conditions that do not cause a state change are not noted. For example, parameter checks and state checks do not cause a state change.
- Additional information describing the verb.

Mapped conversation verbs are preceded by an **MC_** designator. For example, the mapped conversation verb **MC_ALLOCATE** corresponds to the basic conversation verb **ALLOCATE**.

Most parameters supplied to and returned by APPC are hexadecimal values. To simplify coding, these values are represented by meaningful symbolic constants, which are established by **#define** statements in the WINAPPC.H header file. For example, the **opcode** (operation code) member of the **mc_send_data** structure used by the **MC_SEND_DATA** verb is the hexadecimal value represented by the symbolic constant AP_M_SEND_DATA. Use only the symbolic constants when writing TPs.

ALLOCATE

The **ALLOCATE** verb is issued by the invoking TP. It allocates a session between the local LU and partner LU and (in conjunction with RECEIVE_ALLOCATE) establishes a conversation between the invoking TP and the invoked TP. After this verb executes successfully, APPC generates a conversation identifier (**conv_id**). The **conv_id** is a required parameter for all other APPC conversation verbs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **ALLOCATE** verb.

```
struct allocate {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned long    conv_type;
    unsigned char    synclevel;
    unsigned char    reserv3[2];
    unsigned char    rtn_ctl;
    unsigned char    reserv4;
    unsigned long    conv_group_id;
    unsigned long    sense_data;
    unsigned char    plu_alias[8];
    unsigned char    mode_name[8];
    unsigned char    tp_name[64];
    unsigned char    security;
    unsigned char    reserv5[11];
    unsigned char    pwd[10];
    unsigned char    user_id[10];
    unsigned short   pip_dlen;
    unsigned char FAR * pip_dptr;
    unsigned char    reserv7;
    unsigned char    fqplu_name[17];
    unsigned char    reserv8[8];
    unsigned long    proxy_user;
    unsigned long    proxy_domain;
    unsigned char    reserv9[16];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_ALLOCATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION. If the AP_EXTD_VCB bit is set, this indicates that an extended version of the verb control block is used. In this case, the **ALLOCATE** structure includes Sync Point support or privileged proxy feature support.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Returned parameter. Identifies the conversation established between the two TPs.

conv_type
Supplied parameter. Used only by **ALLOCATE** to specify the type of conversation to allocate and is either AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION.

If **ALLOCATE** establishes a mapped conversation, the local TP must issue basic-conversation verbs and provide its own mapping layer to convert data records to logical records and logical records to data records. The partner TP can issue basic-conversation verbs and provide the mapping layer, or it can use mapped-conversation verbs (if the partner TP is using an implementation of APPC that supports mapped-conversation verbs). For more information, see your IBM SNA manual(s).

synclevel
Supplied parameter. Specifies the synchronization level of the conversation. It determines whether the TPs can request confirmation of receipt of data and confirm receipt of data.

- AP_NONE specifies that confirmation processing will not be used in this conversation.
- AP_CONFIRM_SYNC_LEVEL specifies that the TPs can use confirmation processing in this conversation.
- AP_SYNCPT specifies that TPs can use Sync Point Level 2 confirmation processing in this conversation.

reserv3
A reserved field.

rtn_ctl
Supplied parameter. Specifies when the local LU, acting on a session request from the local TP, should return control to the local TP. For information about sessions, see About Transaction Programs.

- AP_IMMEDIATE specifies that the LU allocates a contention-winner session, if one is immediately available, and returns control to the TP.
- AP_WHEN_SESSION_ALLOCATED specifies that the LU does not return control to the TP until it allocates a session or encounters one of the errors documented in Return Codes in this topic. (If the session limit is zero, the LU returns control immediately.) If a session is not available, the TP waits for one.
- AP_WHEN_SESSION_FREE specifies that the LU allocates a contention-winner or contention-loser session, if one is available or able to be activated, and returns control to the TP. If an error occurs, (as documented in Return Codes in this topic) the call will return immediately with the error in the **primary_rc** and **secondary_rc** fields.
- AP_WHEN_CONWINNER_ALLOCATED specifies that the LU does not return control until it allocates a contention-winner session or encounters one of the errors documented in Return Codes in this topic. (If the session limit is zero, the LU returns control immediately.) If a session is not available, the TP waits for one.
- AP_WHEN_CONV_GROUP_ALLOCATED specifies that the LU does not return control to the TP until it allocates the session specified by **conv_group_id** or encounters one of the errors documented in Return Codes in this topic. If a session is not available, the TP waits for it to become free.

Note that AP_IMMEDIATE is the only value for **rtn_ctl** that will never cause a new session to start. For values other than AP_IMMEDIATE, if an appropriate session is not immediately available, Microsoft® Host Integration Server or SNA Server will try to start one. This will cause the on-demand connection to be activated.

reserv4
A reserved field.

conv_group_id
Supplied/returned parameter. Specifies the identifier of the conversation group from which the session should be allocated. The **conv_group_id** is required only if **rtn_ctl** is set to WHEN_CONV_GROUP_ALLOC. When **rtn_ctl** specifies a different value and the **primary_rc** is AP_OK, this is a returned value.

sense_data
Returned parameter. Indicates an allocation error (retry or no-retry) and contains sense data.

plu_alias
Supplied parameter. Specifies the alias by which the partner LU is known to the local TP.

The **plu_alias** must match the name of a partner LU established during configuration.

The parameter is an 8-byte ASCII character string. It can consist of the following ASCII characters:

- Uppercase letters
- Numerals 0 through 9
- Spaces
- Special characters $, #, %, and @

The first character of this string cannot be a space.

If the value of this parameter is fewer than eight bytes, pad it on the right with ASCII spaces (0x20).

If you want to specify the partner LU with the **fqplu_name** parameter, fill this parameter with binary zeros.

For a user or group using TPs, 5250 emulators, and/or APPC applications, the system administrator can assign default local and remote LUs. In this case, the field is left blank or null and the default LUs are accessed when the user or group member starts an APPC program. For more information on default LUs, see the *Microsoft Host Integration Server 2000 online books*.

mode_name

Supplied parameter. Specifies the name of a set of networking characteristics defined during configuration.

The value of **mode_name** must match the name of a mode associated with the partner LU during configuration.

The parameter is an 8-byte EBCDIC character string. It can consist of characters from the type A EBCDIC character set:

- Uppercase letters
- Numerals 0 through 9
- Special characters $, #, and @

The first character in the string must be an uppercase letter or a special character.

Do not use SNASVCMG in a mapped conversation. SNASVCMG is a reserved **mode_name** used internally by APPC. Using this name in a basic conversation is not recommended.

tp_name

Supplied parameter. Specifies the name of the invoked TP. The value of **tp_name** specified by **ALLOCATE** in the invoking TP must match the value of **tp_name** specified by RECEIVE_ALLOCATE in the invoked TP.

The parameter is a 64-byte EBCDIC character string and is case-sensitive. The **tp_name** parameter can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If **tp_name** is fewer than 64 bytes, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention is that a service TP name can have up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The other characters are from the type AE EBCDIC character set.

security

Supplied parameter. Provides the information that the partner LU requires to validate access to the invoked TP.

Based on the conversation security established for the invoked TP during configuration, use one of the following values:

- AP_NONE for an invoked TP that uses no conversation security.
- AP_PGM for an invoked TP that uses conversation security and thus requires a user identifier and password. Supply this information through the **user_id** and **pwd** parameters.
- AP_PROXY_PGM for an invoked TP with privileged proxy that uses conversation security and thus requires a user identifier and password. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields.
- AP_PROXY_SAME for a TP that has been invoked using privileged proxy with a valid user identifier and password supplied by the proxy, which in turn invokes another TP. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields.

For example, assume that TP A invokes TP B with a valid user identifier and password supplied by the privileged proxy, and TP B in turn invokes TP C. If TP B specifies the value AP_PROXY_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator tells TP C to not require the password (if TP C is configured to accept an already-verified indicator).

- AP_PROXY_STRONG for an invoked TP with privileged proxy that uses conversation security and thus requires a user identifier and password provided by the privileged proxy mechanism. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields. AP_PROXY_STRONG differs from AP_PROXY_PGM in

that AP_PROXY_STRONG does not allow clear-text passwords. If the remote system does not support encrypted passwords (strong conversation security), then this call fails.

- AP_SAME for a TP that has been invoked with a valid user identifier and password, which in turn invokes another TP.

For example, assume that TP A invokes TP B with a valid user identifier and password, and TP B in turn invokes TP C. If TP B specifies the value AP_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator tells TP C to not require the password (if TP C is configured to accept an already-verified indicator).

When AP_SAME is used in an Allocate verb, your application must always provide values for the **user_id** and **pwd** parameters in the verb control block. Depending on the properties negotiated between the SNA server and the peer LU, the **ALLOCATE** verb will send one of 3 kinds of Attach (FMH-5) messages, in this order of precedence:

1. If the LUs have negotiated "already verified" security, then the Attach sent by the SNA server will not include the contents of the **pwd** parameter field specified in the VCB.
2. If the LUs have negotiated "persistent verification" security, then the Attach sent by the SNA server will include the **pwd** parameter specified in the VCB, but only when the Attach is the first for the specified **user_id** parameter since the start of the LU-LU session, and will omit the **pwd** parameter on all subsequent Attaches (issued by your application or any other application using this LU-LU-mode triplet).
3. If the LUs have not negotiated either of the above, then the Attach sent by the SNA server will omit both the **user_id** and **pwd** parameters on all Attaches.

   Your application cannot tell which mode of security has been negotiated between the LUs, nor can it tell whether the **ALLOCATE** verb it is issuing is the first for that LU-LU-mode triplet. So your application must always set the **user_id** and **pwd** parameter fields in the VCB when **security** is set to AP_SAME.

   For more information on persistent verification and already verified security, see the SNA Formats Guide, section "FM Header 5: Attach (LU 6.2)".

- AP_STRONG for an invoked TP that uses conversation security and thus requires a user identifier and password. Supply this information through the **user_id** and **pwd** parameters. AP_STRONG differs from AP_PGM in that AP_STRONG does not allow clear-text passwords. If the remote system does not support encrypted passwords (strong conversation security), then this call fails.

  If the APPC automatic logon feature is to be used, **security** must be set to AP_PGM. See the Remarks section for details.

reserv5
A reserved field.

pwd
Supplied parameter. Specifies the password associated with **user_id**.

The **pwd** parameter is required only if **security** is set to AP_PGM or AP_SAME. It must match the password for **user_id** that was established during configuration.

The **pwd** parameter is a 10-byte EBCDIC character string and is case-sensitive. It can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If the password is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

If the APPC automatic logon feature is to be used, the **pwd** character string must be hard-coded to MS$SAME. See the Remarks section for details.

user_id
Supplied parameter. Specifies the user identifier required to access the partner TP. It is required only if the security parameter is set to AP_PGM or AP_SAME.

The **user_id** parameter is a 10-byte EBCDIC character string and is case-sensitive. It must match one of the user identifiers configured for the partner TP.

The parameter can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If **user_id** is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

If the APPC automatic logon feature is to be used, the **user_id** character string must be hard-coded to MS$SAME. See the Remarks section for details.

pip_dlen
Supplied parameter. Specifies the length of the program initialization parameters (PIP) to be passed to the partner TP. The range is from 0 through 32767.

pip_dptr
Supplied parameter. Specifies the address of the buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

PIP data can consist of initialization parameters or environmental setup information required by a partner TP or remote operating system. The PIP data must follow the general data stream (GDS) format. For more information, see *SNA LU6.2 Reference: Peer Protocols* published by IBM.

For Microsoft® Windows NT®, Windows 95, Windows 98, and Windows version 3.*x*, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For OS/2, the PIP data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The PIP data buffer must fit entirely on the segment.

reserv7
A reserved field.

fqplu_name
Supplied parameter. Specifies the fully qualified name of the partner LU. This must match the fully qualified name of the local LU defined in the remote node. The parameter consists of two type A EBCDIC character strings for the NETID and the LU name of the partner LU. The names are separated by an EBCDIC period (.).

This name must be provided if no **plu_alias** is specified. It can consist of the following EBCDIC characters:

- Uppercase letters
- Numerals 0 through 9
- Special characters $, #, and @

If the value of this parameter is fewer than 17 bytes, pad it on the right with EBCDIC spaces (0x40).

reserv8
A reserved field.

proxy_user
Supplied parameter. Specifies a LPWSTR pointing to a UNICODE string containing the user name to be impersonated using the privileged proxy feature. This field can only be used when the AP_EXTD_VCB bit is set on the **opext** field, indicating an extended VCB.

proxy_domain
Supplied parameter. Specifies a LPWSTR pointing to a UNICODE string containing the domain name of the user to be impersonated using the privileged proxy feature. This field can only be used when the AP_EXTD_VCB bit is set on the **opext** field, indicating an extended VCB.

reserv9
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL
Primary return code; the supplied parameter **rtn_ctl** specified immediate (AP_IMMEDIATE) return of control to the TP, and the local LU did not have an available contention-winner session.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_RETURN_CONTROL

Secondary return code; the value specified for **rtn_ctl** was invalid.

AP_BAD_SECURITY

Secondary return code; the value specified for **security** was invalid.

AP_BAD_SYNC_LEVEL

Secondary return code; the value specified for **sync_level** was invalid.

AP_BAD_TP_ID

Secondary return code; the value specified for **tp_id** was invalid.

AP_PIP_LEN_INCORRECT

Secondary return code; the value of **pip_dlen** was greater than 32767.

AP_UNKNOWN_PARTNER_MODE

Secondary return code; the value specified for **mode_name** was invalid.

AP_BAD_PARTNER_LU_ALIAS

Secondary return code; APPC did not recognize the supplied **partner_lu_alias**.

AP_BAD_CONV_TYPE (for a basic conversation)

Secondary return code; the value specified for **conv_type** was invalid.

AP_NO_USE_OF_SNASVCMG (for a mapped conversation)

Secondary return code; SNASVCMG is not a valid value for **mode_name**.

AP_INVALID_DATA_SEGMENT

Secondary return code; the PIP data was longer than the allocated data segment, or the address of the PIP data buffer was wrong.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after **ALLOCATE**.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **ALLOCATE**, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for system configured with multiple nodes using Host Integration Server or SNA Server, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

ALLOCATE can establish either a basic or mapped conversation.

The conversation state is RESET when the TP issues this verb. After successful execution (**primary_rc** is AP_OK), the state changes to SEND. If the verb does not execute, the state remains unchanged.

Several parameters of **ALLOCATE** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

To send the **ALLOCATE** request immediately, the invoking TP can issue FLUSH or CONFIRM immediately after **ALLOCATE**. Otherwise, the **ALLOCATE** request accumulates with other data in the local LU's send buffer until the buffer is full.

By issuing **CONFIRM** after **ALLOCATE**, the invoking TP can immediately determine whether the allocation was successful (if **synclevel** is set to AP_CONFIRM_SYNC_LEVEL).

Normally, the value of the **ALLOCATE** verb's **mode_name** parameter must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU.

If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**.

Host Integration Server and SNA Server support a feature called password substitution. This is a security feature supported by the latest version of the OS/400 operating system (V3R1) which encrypts any password that flows between two nodes on an Attach message. A password flows on an Attach whenever someone invokes an APPC transaction program specifying a user identifier and password. For example, this happens whenever anyone logs on to an AS/400.

Support for password substitution is indicated by setting bit 5 in byte 23 of the BIND request to 1 (which indicates that password substitution is supported). If the remote system sets this bit in the BIND response, the SNA server automatically encrypts the LU 6.2 conversation security password included in the FMH-5 Attach message. APPC applications using Host Integration Server or SNA Server automatically take advantage of this feature by setting the **security** field of the VCB to AP_PGM or AP_STRONG in the **ALLOCATE** request.

If an APPC application wants to force an encrypted password to flow, the application can specify AP_STRONG for the **security** field in the VCB in the **ALLOCATE** request. This option is implemented as defined in OS/400 V3R1, and is documented in the OS/400 CPIC programmer reference as CM_SECURITY_PROGRAM_STRONG, where the LU 6.2 **pwd** (password) field is encrypted before it flows over the physical network.

The password substitution features is currently only supported by OS/400 V3R1 or later. If the remote system does not support this feature, the SNA server will UNBIND the session with the sense code of 10060006. The two nodes negotiate whether or not they support this feature in the BIND exchange. Host Integration Server and SNA Server set a bit in the BIND, and also adds some random data on the BIND for encryption. If the remote node supports password substitution, it sets the same bit in the BIND response, and adds some (different) random data for decryption.

Host Integration Server 2000, SNA Server version 4.0, and SNA Server version 3.0 with Service Pack 1 or later and support

automatic logon for APPC applications. This feature requires specific configuration by the network administrator: The APPC application must be invoked on the LAN side from a client of Host Integration Server 2000. The client must be logged into a Windows 2000 or Windows NT domain, but the client can be running on any operating system supported by the Host Integration Server or SNA Server APPC APIs.

The client application is coded to use "program" level security, with a special hard-coded APPC user name MS$SAME and password MS$SAME. When this session allocation flows from client to SNA server, the server looks up the host account and password corresponding to the Windows 2000 or Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

It is illegal for the remote node to set the bit specifying password substitution and not add the random data.

According to IBM, there are implementations of LU 6.2 password substitution that do not support password substitution but do echo the password substitution bit back to Host Integration Server 2000, without specifying any random data. When they do this, the SNA server will UNBIND the session with the sense code 10060006.This sense code is interpreted as:

- 1006 = Required field or parameter missing.
- 0006 = A required subfield of a control vector was omitted.

Host Integration Server and SNA Server should also log an Event 17 (APPC session activation failure: BIND negative response sent).

The correct solution is for the failing implementation to be fixed. However, as a short-term workaround, the following Host Integration Server or SNA Server service registry setting can be set:

HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\snaservr\parameters\NOPWDSUB: REG_SZ: YES

When this parameter is specified in the registry, password substitution support will be disabled.

Several updates have been made to Host Integration Server and SNA Server to allow a privileged APPC application to open an APPC conversation using the Single Signon feature on behalf of any defined Windows NT user. This is referred to as the privileged proxy feature. An extension has been added to the APPC **ALLOCATE** verb to invoke this feature.

An APPC application becomes privileged by being started in a Windows NT user account that is a member of a special Windows NT group. When a Host Security Domain is configured, Host Integration Server 2000 Manager will define a second Windows NT group for use with the host security features of Host Integration Server 2000. If the user account under which the actual client is running is a member of this second Windows NT group, the client is privileged to initiate an APPC conversation on behalf of any user account defined in the Host Account Cache.

The following illustrates how the privileged proxy feature works:

The Host Integration Server 2000 administrator creates a Host Security Domain called APP. Host Integration Server 2000 Manager now creates two Windows NT groups. The first group is called APP and the second is called APP_PROXY for this example. Users that are assigned to the APP group are enabled for single signon. Users assigned to the APP_PROXY group are privileged proxies. The administrator adds the Windows NT user AppcUser to the APP_PROXY group using the Users button on the Host Security Domain property dialog box in Host Integration Server 2000 Manager.

The administrator then sets up an APPC application on the Host Integration Server 2000 to run as a Windows NT service called APPCAPP and that service has been setup to operate under the AppcUser user account. When APPCAPP runs, it opens an APPC session via an ALLOCATE verb using the extended VCB format and specifies the Windows NT username of the desired user, UserA (for example).

The SNA service sees the session request coming from a connection that is a member of the Host Security Domain APP. The Client/Server interface tells the SNA service that the actual client is AppcUser.

The SNA service checks to see if AppcUser is a member of the APP_PROXY group. Because AppcUser is a member of APP_PROXY, the SNA service inserts the Username/Password for UserA in the APPC Attach (FMH-5) command and sends it off to the partner TP.

In order to support the privileged proxy feature, the APPC application must implement the following program logic:

The APPC application must determine the Windows NT user ID and domain name that it wishes to impersonate.

The APPC application must set the following parameters before calling the **ALLOCATE** verb:

Enable the use of the extended ALLOCATE verb control block structure by setting the AP_EXTD_VCB flag in the **opext** field.

Set **security** to AP_PROXY_SAME, AP_PROXY_PGM or AP_PROXY_STRONG.

Set up the pointers for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name

of the user to be impersonated.

> **Note** The application does not need to set up the **user_id** and **pwd** fields in the **ALLOCATE** VCB.

When the APPC application performs the above steps and issues the **ALLOCATE** verb, the Host Integration Server 2000 server will perform a lookup in the host security domain for the specified Windows NT user and set the user ID and password fields in the FMH-5 Attach message sent to the remote system.

CONFIRM

The **CONFIRM** verb sends the contents of the local LU's send buffer and a confirmation request to the partner TP.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **CONFIRM** verb.

```
struct confirm {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned long   conv_id;
    unsigned char   rts_rcvd;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_CONFIRM.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Returned parameter. Identifies the conversation established between the two TPs.

rts_rcvd
Returned parameter. Indicates whether the partner TP issued REQUEST_TO_SEND which requests the local TP to change the conversation to RECEIVE state.

To change to RECEIVE state operating on Microsoft® Windows NT®, Windows 95, or Windows 98, the local TP can use PREPARE_TO_RECEIVE, RECEIVE_AND_WAIT, or RECEIVE_AND_POST.

To change to RECEIVE state operating on Windows 3.x, the local TP can use **PREPARE_TO_RECEIVE** or **RECEIVE_AND_WAIT**.

To change to RECEIVE state operating on OS/2, the local TP can use **RECEIVE_AND_POST**.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_CONFIRM_ON_SYNC_LEVEL_NONE

Secondary return code; the local TP attempted to use **CONFIRM** in a conversation with a synchronization level of AP_NONE.

The synchronization level, established by ALLOCATE, must be AP_CONFIRM_SYNC_LEVEL.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_CONFIRM_BAD_STATE

Secondary return code; the conversation was not in SEND state.

AP_CONFIRM_NOT_LL_BDY

Secondary return code; the conversation for the local TP was in SEND state, and the local TP did not finish sending a logical record.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after **ALLOCATE**.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE, AP_CONFIRM_SYNC_LEVEL, or AP_SYNCPT) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.

- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer has encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued SEND_ERROR with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, Windows 98, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

Remarks

In response to **CONFIRM**, the partner TP normally issues CONFIRMED to confirm that it has received the data without error. (If the partner TP encounters an error, it issues SEND_ERROR or abnormally deallocates the conversation.)

The TP can issue **CONFIRM** only if the conversation's synchronization level, established by ALLOCATE, is AP_CONFIRM_SYNC_LEVEL.

The conversation must be in SEND state when the TP issues this verb. State changes, summarized in the following table, are based on the value of the **primary_rc**.

primary_rc	**New state**
AP_OK	No change
AP_ALLOCATION_ERROR	RESET

AP_COMM_SUBSYSTEM_ABENDED AP_COMM_SUBSYSTEM_NOT_LOADED	RESET RESET
AP_CONV_FAILURE_RETRY AP_CONV_FAILURE_NO_RETRY	RESET RESET
AP_DEALLOC_ABEND AP_DEALLOC_ABEND_PROG AP_DEALLOC_ABEND_SVC AP_DEALLOC_ABEND_TIMER	RESET RESET RESET RESET
AP_PROG_ERROR_PURGING AP_SVC_ERROR_PURGING	RECEIVE RECEIVE

CONFIRM waits for a response from the partner TP. A response is generated by one of the following verbs in the partner TP:

- CONFIRMED
- SEND_ERROR
- DEALLOCATE with dealloc_type set to AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER
- TP_ENDED

By issuing **CONFIRM** after ALLOCATE, the invoking TP can immediately determine whether the allocation was successful (if **synclevel** is set to AP_CONFIRM_SYNC_LEVEL).

Normally, the value of the **ALLOCATE** verb's **mode_name** parameter must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU.

If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**. For more information, see the *Microsoft Host Integration Server 2000 online books*.

Several parameters of **ALLOCATE** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

To send the **ALLOCATE** request immediately, the invoking TP can issue FLUSH or **CONFIRM** immediately after **ALLOCATE**. Otherwise, the **ALLOCATE** request accumulates with other data in the local LU's send buffer until the buffer is full.

CONFIRMED

The **CONFIRMED** verb responds to a confirmation request from the partner TP. It informs the partner TP that the local TP has not detected an error in the received data. Because the TP issuing the confirmation request waits for a confirmation, **CONFIRMED** synchronizes the processing of the two TPs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **CONFIRMED** verb.

```
struct confirmed {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned long   conv_id;
    unsigned char   rts_rcvd;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_CONFIRMED.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Identifies the conversation established between the two TPs. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter. Indicates whether the partner TP issued MC_REQUEST_TO_SEND, which requests the local TP to change the conversation to RECEIVE state.

To change to RECEIVE state the local TP can use MC_PREPARE_TO_RECEIVE, MC_RECEIVE_AND_WAIT, or MC_RECEIVE_AND_POST.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_CONFIRMED_BAD_STATE

Secondary return code; the conversation is not in CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY

Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING

Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation must be in one of the following states when the TP issues this verb:

- CONFIRM
- CONFIRM_SEND
- CONFIRM_DEALLOCATE

The new state is determined by the old state—the state of the conversation when the local TP issued **CONFIRMED**. The old state is indicated by the value of the **what_rcvd** parameter of the preceding receive verb. The following state changes are possible:

Old state	New state

CONFIRM	RECEIVE
CONFIRM_SEND	SEND
CONFIRM_DEALLOCATE	RESET

Confirmation Requests

A confirmation request is issued by one of the following verbs in the partner TP:

- CONFIRM
- PREPARE_TO_RECEIVE if **ptr_type** is set to AP_SYNC_LEVEL and the conversation's synchronization level (established by ALLOCATE) is AP_CONFIRM_SYNC_LEVEL
- DEALLOCATE if **dealloc_type** is set to AP_SYNC_LEVEL and the conversation's synchronization level (established by **ALLOCATE**) is AP_CONFIRM_SYNC_LEVEL
- SEND_DATA if type is set to AP_SEND_DATA_CONFIRM and the conversation's synchronization level (established by **ALLOCATE**) is AP_CONFIRM_SYNC_LEVEL

A confirmation request is received by the local TP through the **what_rcvd** parameter of one of the following verbs:

- RECEIVE_IMMEDIATE
- RECEIVE_AND_WAIT
- RECEIVE_AND_POST

CONFIRMED is issued by the local TP only if **what_rcvd** contains one of the following values:

- AP_CONFIRM_WHAT_RECEIVED
- AP_CONFIRM_SEND
- AP_CONFIRM_DEALLOCATE

If the **rtn_status** parameter is set to AP_YES, **what_rcvd** can also contain the following values:

- AP_DATA_COMPLETE_CONFIRM
- AP_DATA_COMPLETE_CONFIRM_SEND
- AP_DATA_COMPLETE_CONFIRM_DEALL

For basic conversations, **what_rcvd** can also contain the following values:

- AP_DATA_CONFIRM
- AP_DATA_CONFIRM_SEND
- AP_DATA_CONFIRM_DEALLOCATE

DEALLOCATE

The **DEALLOCATE** verb deallocates a conversation between two TPs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC function rather than the blocking version of this call.

The following structure describes the verb control block used by the **DEALLOCATE** verb.

```
struct deallocate {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        tp_id[8];
    unsigned long        conv_id;
    unsigned char        reserv3;
    unsigned char        dealloc_type;
    unsigned short       log_dlen;
    unsigned char FAR *  log_dptr;
    void                 (WINAPI *callback)();
    void                 *correlator;
    unsigned char        reserv4[4];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_DEALLOCATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Identifies the conversation established between the two TPs. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

dealloc_type
Supplied parameter. Specifies how to perform the deallocation.

Using one of the following values deallocates the conversation abnormally:

- AP_ABEND_PROG
- AP_ABEND_SVC
- AP_ABEND_TIMER

If the conversation is in SEND state when the local TP issues **DEALLOCATE**, APPC sends the contents of the local LU's send buffer to the partner TP before deallocating the conversation. If the conversation is in RECEIVE or PENDING_POST state, APPC purges any incoming data before deallocating the conversation.

An application or service TP should specify AP_ABEND_PROG when it encounters an error preventing the successful completion of a transaction.

A service TP should specify AP_ABEND_SVC when it encounters an error caused by its partner service TP (for example, a format error in control information sent by the partner service TP). A service TP should specify AP_ABEND_TIMER when it encounters an error requiring immediate deallocation (for example, an operator ending the program prematurely).

AP_FLUSH sends the contents of the local LU's send buffer to the partner TP before deallocating the conversation. This value is allowed only if the conversation is in SEND state.

AP_SYNC_LEVEL uses the conversation's synchronization level (established by ALLOCATE) to determine how to deallocate the conversation. This value is allowed only if the conversation is in SEND state.

If the synchronization level of the conversation is AP_NONE, APPC sends the contents of the local LU's send buffer to the partner TP before deallocating the conversation.

If the synchronization level is AP_CONFIRM_SYNC_LEVEL, APPC sends the contents of the local LU's send buffer and a confirmation request to the partner TP. Upon receiving confirmation from the partner TP, APPC deallocates the conversation. If, however, the partner TP reports an error, the conversation remains allocated.

log_dlen

Supplied parameter. Specifies the number of bytes of data to be sent to the error log file. The range is from 0 through 32767.

You can set this parameter to a number greater than zero if **dealloc_type** is set to AP_ABEND_PGM, AP_ABEND_SVC, or AP_ABEND_TIMER. Otherwise, this parameter must be zero.

log_dptr

Supplied parameter. Provides the address of the data buffer containing error information. The data is sent to the local error log and to the partner LU.

This parameter is used by **DEALLOCATE** if **log_dlen** is greater than zero.

For Microsoft® Windows NT®, Windows 95, Windows 98, and Windows 3.x, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For OS/2, the log data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The log data buffer must fit entirely on the segment.

The TP must format the error data as a GDS error log variable. For more information, see your IBM SNA manual(s).

callback

Supplied parameter. Only present if the AP_EXTD_VCB bit is set in the **opext** member, indicating support for Sync Point. This parameter is the address of a user-supplied callback function. If this field is NULL, no notification will be provided.

The prototype of the callback routine is as follows:

```
void WINAPI callback_proc(
struct appc_hdr *vcb,
unsigned char tp_id[8],
unsigned long conv_id,
unsigned short type,
void *correlator
);
```

The callback procedure can take any name, since the address of the procedure is passed to the APPC DLL. The parameters passed to the function are as follows:

vcb

A pointer to the **DEALLOCATE** verb control block that caused the conversation to be deallocated.

tp_id

The TP identifier of the TP that owned the deallocated conversation.

conv_id

The conversation identifier of the deallocated conversation.

type

The type of the message flow that caused the callback to be invoked. Possible values are:

AP_DATA_FLOW

Normal data flow on the session.

AP_UNBIND

The session was unbound normally.

AP_FAILURE

The session terminated due to an outage.

correlator

This value is the **correlator** specified on the **DEALLOCATE** verb.

correlator
Supplied parameter. Only present if the AP_EXTD_VCB bit is set in the **opext** member, indicating support for the Sync Point API. This correlator field allows the TP to specify a value it can use to correlate a call to the callback function with, for example, its own internal data structures. This value is returned to the TP as one of the parameters of the callback routine when it is invoked.

reserv4
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_DEALLOC_BAD_TYPE

Secondary return code; the **dealloc_type** parameter was not set to a valid value.

AP_DEALLOC_LOG_LL_WRONG

Secondary return code; the LL field of the GDS error log variable did not match the actual length of the log data.

AP_INVALID_DATA_SEGMENT

Secondary return code; the error data for the log file was longer than the segment allocated to contain the error data, or the address of the error data buffer was wrong.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_DEALLOC_CONFIRM_BAD_STATE

Secondary return code; the conversation was not in SEND state, and the TP attempted to flush the send buffer and send a confirmation request. This attempt occurred because the value of **dealloc_type** was AP_SYNC_LEVEL and the synchronization level of the conversation was AP_CONFIRM_SYNC_LEVEL.

AP_DEALLOC_FLUSH_BAD_STATE

Secondary return code; the conversation was not in SEND state and the TP attempted to flush the send buffer. This attempt occurred because the value of **dealloc_type** was AP_FLUSH or because the value of **dealloc_type** was AP_SYNC_LEVEL and the synchronization level of the conversation was AP_NONE. In either case, the conversation must be in SEND state.

AP_DEALLOC_NOT_LL_BDY

Secondary return code; the conversation was in SEND state, and the TP did not finish sending a logical record. The **dealloc_type** parameter was set to AP_SYNC_LEVEL or AP_FLUSH.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the

allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for an Host Integration Server 2000 system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued SEND_ERROR with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, Windows 98, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

Remarks

Depending on the value of the **dealloc_type** parameter, the conversation can be in one of the states indicated in the following table when the TP issues **DEALLOCATE**.

dealloc_type	**Allowed state**
AP_FLUSH	SEND
AP_SYNC_LEVEL	SEND
AP_ABEND	Any except RESET
AP_ABEND_PROG	Any except RESET
AP_ABEND_SVC	Any except RESET

AP_ABEND_TIMER	Any except RESET

State changes, summarized in the following table, are based on the value of the **primary_rc**.

primary_rc	New state
AP_OK	RESET
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

Before deallocating the conversation, this verb performs the equivalent of one of the following:

- FLUSH, by sending the contents of the local LU's send buffer to the partner LU (and TP).
- CONFIRM, by sending the contents of the local LU's send buffer and a confirmation request to the partner TP.

After this verb has successfully executed, the conversation identifier is no longer valid.

LU 6.2 Sync Point can use an optimization of the message flows known as implied forget. When the protocol specifies that a FORGET PS header is required, the next data flow on the session implies that a FORGET has been received. In the normal situation, the TP is aware of the next data flow when data is received or sent on one of its Sync Point conversations.

However, it is possible that the last message to flow is caused by the conversation being deallocated. In this case, the TP is unaware when the next data flow on the session occurs. To provide the TP with this notification, the **DEALLOCATE** verb is modified to allow the TP to register a callback function which will be called:

- On the first normal flow transmission (request or response) over the session used by the conversation.
- If the session is unbound before any other data flows.
- If the session is terminated abnormally due to a DLC outage.

The **DEALLOCATE** verb also contains a correlator field member that is returned as one of the parameters when the callback function is invoked. The application can use this parameter in any way (for example, as a pointer to a control block within the application).

The TP can use the *type* parameter passed to the callback function to determine whether the message flow indicates an implied forget has been received.

Note that the **DEALLOCATE** verb will probably complete before the callback routine is called. The conversation is considered to be in RESET state and no further verbs can be issued using the conversation identifier. If the application issues a TP_ENDED verb before the next data flow on the session, the callback routine will not be invoked.

Host Integration Server 2000 allows TPs to deallocate conversations immediately after sending data by specifying the **type** parameter on SEND_DATA as AP_SEND_DATA_DEALLOC_*. However, the **SEND_DATA** verbs do not contain the implied forget callback function. TPs wishing to receive implied forget notification must issue **DEALLOCATE** explicitly.

FLUSH

The **FLUSH** verb sends the contents of the local LU's send buffer to the partner LU (and TP). If the send buffer is empty, no action takes place.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **FLUSH** verb.

```
struct flush {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_FLUSH.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_FLUSH_NOT_SEND_STATE

Secondary return code; the conversation was not in SEND state.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.

- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

Data processed by SEND_DATA accumulates in the local LU's send buffer until one of the following happens:

- The local TP issues **FLUSH** (or other verb that flushes the LU's send buffer).
- The buffer is full.

The request generated by ALLOCATE is also buffered.

The conversation must be in SEND state when the TP issues this verb.

There is no state change.

GET_ATTRIBUTES

The **GET_ATTRIBUTES** verb returns the attributes of the conversation.

The following structure describes the verb control block used by the **GET_ATTRIBUTES** verb.

```
struct get_attributes {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    reserv3;
    unsigned char    sync_level;
    unsigned char    mode_name[8];
    unsigned char    net_name[8];
    unsigned char    lu_name[8];
    unsigned char    lu_alias[8];
    unsigned char    plu_alias[8];
    unsigned char    plu_un_name[8];
    unsigned char    reserv4[2];
    unsigned char    fqplu_name[17];
    unsigned char    reserv5;
    unsigned char    user_id[10];
    unsigned long    conv_group_id;
    unsigned char    conv_corr_len;
    unsigned char    conv_corr[8];
    unsigned char    reserv6[13];
NOTE: The following fields are present when the high bit of opext is set (opext & AP_EXTD_VCB
) != 0.
    unsigned char    luw_id[26];
    unsigned char    sess_id[8];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_GET_ATTRIBUTES.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

Reserv3
A reserved field.

sync_level
Returned parameter. Specifies the level of synchronization processing for the conversation. This parameter determines whether the TPs can request confirmation of receipt of data and confirm receipt of data.

- AP_NONE indicates that confirmation processing will not be used in this conversation.
- AP_CONFIRM_SYNC_LEVEL indicates that TPs can use confirmation processing in this conversation.

- AP_SYNCPT indicates that TPs can use Sync Point Level 2 confirmation processing in this conversation.

mode_name
Returned parameter. Specifies the name of a set of networking characteristics. It is a type A EBCDIC character string.
net_name
Returned parameter. Specifies the name of the SNA network containing the local LU used by this TP. It is a type A EBCDIC character string.
lu_name
Returned parameter. Provides the name of the local LU.
lu_alias
Returned parameter. Provides the alias by which the local LU is known to the local TP. It is an ASCII character string.
plu_alias
Returned parameter. Provides the alias by which the partner LU is known to the local TP. It is an ASCII character string.
plu_un_name
Returned parameter. Specifies the uninterpreted name of the partner LU — the name of the partner LU as defined to the system services control point (SSCP). It is a type AE EBCDIC character string. This parameter is returned only if the local LU is dependent.
Reserv4
A reserved field.
fqplu_name
Returned parameter. Provides the fully qualified name of the partner LU. It is a type A EBCDIC character string. The field contains the network name, an EBCDIC period, and the partner-LU name.
Reserv5
A reserved field.
user_id
Returned parameter. Specifies the user identifier sent by the invoking TP through ALLOCATE to access the invoked TP (if applicable). It is a type AE EBCDIC character string. The field contains the user identifier if the following conditions are true:

- The invoked TP requires conversation security.
- **GET_ATTRIBUTES** was issued by the invoked TP.

Otherwise, the field contains spaces.

conv_group_id
Returned parameter. Specifies the conversation group identifier for the session to which the conversation has been allocated. This is also returned on ALLOCATE and RECEIVE_ALLOCATE.
Conv_Corr_len
Returned parameter. Specifies the length of the conversation correlator identifier that is returned.
Conv_corr
Returned parameter. Specifies the conversation correlator identifier (if any) that the source LU assigns to identify the conversation, which is unique for the source/partner LU pair. It is sent by the source LU on the allocation request.

The following fields are present when the high bit of **opext** is set (**opext** & AP_EXTD_VCB) != 0.These fields are only present when when using Sync Point Level 2 support.

Reserv6
A reserved field.
Luw_id
Logical unit-of-work identifier.
Sess_id
Session identifier.

Return Codes

AP_OK
Primary return code; the verb executed successfully.
AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

GET_LU_STATUS

The **GET_LU_STATUS** verb returns the status of a particular LU. This conversation verb is only available when Sync Point conversations are supported.

The following structure describes the verb control block used by the **GET_LU_STATUS** verb.

```
struct get_type {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned char    plu_alias[8];
    unsigned short   active_sess;
    unsigned char    zero_sess;
    unsigned char    reserv3[7];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_GET_LU_STATUS.
opext
This field is unused by the **GET_LU_STATUS** verb.
reserv2
A reserved field.
primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.
secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.
tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.
plu_alias
Supplied parameter. Provides the identifier for the LU about which this TP is inquiring. The value of this parameter was returned by MC_ALLOCATE or ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.
active_sess
Returned parameter. Supplies the number of active sessions on this LU.
zero_sess
Returned parameter. Indicates whether a zero session is on this LU. Values are AP_YES or AP_NO.
reserv3
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.
AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_LU_ALIAS

Secondary return code; the value of **plu_alias** did not match any LUs assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

GET_STATE

The **GET_STATE** verb returns the state of a particular conversation.

The following structure describes the verb control block used by the **GET_STATE** verb.

```
struct get_state {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    conv_state;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_GET_STATE.

opext
This field is unused by the **GET_STATE** verb.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the identifier for the conversation about which this TP is inquiring. The value of this parameter was returned by MC_ALLOCATE or ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

conv_state
Returned parameter. Indicates the state of the conversation. The **conv_state** parameter can be one of the following values:

AP_RESET_STATE

The conversation is in the RESET state.

AP_SEND_STATE

The conversation is in the SEND state.

AP_RECEIVE_STATE

The conversation is in the RECEIVE state.

AP_CONFIRM_STATE

The conversation is in the CONFIRM state.

AP_CONFIRM_SEND_STATE

The conversation is in the CONFIRM_SEND state.

AP_CONFIRM_DEALL_STATE

The conversation is in the CONFIRM_DEALLOCATE state.

AP_PEND_POST_STATE

The conversation has a POST verb pending.

AP_PEND_DEALL_STATE

The conversation has a DEALLOCATE verb pending.

AP_END_CONV_STATE

The conversation is in the END_CONVERSATION state.

AP_SEND_PENDING_STATE

The conversation is in the SEND_PENDING state.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state when the TP issues this verb.

There is no state change.

GET_TYPE

The **GET_TYPE** verb returns the conversation type (basic or mapped) of a particular conversation so the TP can decide whether to use basic or mapped conversation verbs.

The following structure describes the verb control block used by the **GET_TYPE** verb.

```
struct get_type {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    conv_type;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_GET_TYPE.

opext
This field is unused by the **GET_TYPE** verb.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the identifier for the conversation about which this TP is inquiring. The value of this parameter was returned by MC_ALLOCATE or ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

conv_type
Returned parameter. Supplies the type of conversation, either AP_BASIC_CONVERSATION or AP_MAPPED_CONVERSATION.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

MC_ALLOCATE

The **MC_ALLOCATE** verb is issued by the invoking TP. It allocates a session between the local LU and partner LU and (in conjunction with RECEIVE_ALLOCATE) establishes a conversation between the invoking TP and the invoked TP. After this verb executes successfully, APPC generates a conversation identifier (**conv_id**). The **conv_id** is a required parameter for all other APPC conversation verbs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_ALLOCATE** verb.

```
struct mc_allocate {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    reserv3;
    unsigned char    synclevel;
    unsigned char    reserv4[2];
    unsigned char    rtn_ctl;
    unsigned char    reserv5;
    unsigned long    conv_group_id;
    unsigned long    sense_data;
    unsigned char    plu_alias[8];
    unsigned char    mode_name[8];
    unsigned char    tp_name[64];
    unsigned char    security;
    unsigned char    reserv6[11];
    unsigned char    pwd[10];
    unsigned char    user_id[10];
    unsigned short   pip_dlen;
    unsigned char FAR * pip_dptr;
    unsigned char    reserv7;
    unsigned char    fqplu_name[17];
    unsigned char    reserv8[8];
    unsigned long    proxy_user;
    unsigned long    proxy_domain;
    unsigned char    reserv9[16];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code; AP_M_ALLOCATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION. If the AP_EXTD_VCB bit is set, this indicates that an extended version of the verb control block is used. In this case, the **MC_ALLOCATE** structure includes Sync Point support or privileged proxy feature support.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Returned parameter. Identifies the conversation established between the two TPs.

reserv3
A reserved field.
synclevel
Supplied parameter. Specifies the synchronization level of the conversation. It determines whether the TPs can request confirmation of receipt of data and confirm receipt of data.

- AP_NONE specifies that confirmation processing will not be used in this conversation.
- AP_CONFIRM_SYNC_LEVEL specifies that the TPs can use confirmation processing in this conversation.
- AP_SYNCPT specifies that TPs can use Sync Point Level 2 confirmation processing in this conversation.

reserv4
A reserved field.
rtn_ctl
Supplied parameter. Specifies when the local LU, acting on a session request from the local TP, should return control to the local TP. For information about sessions, see About Transaction Programs.

- AP_IMMEDIATE specifies that the LU allocates a contention-winner session, if one is immediately available, and returns control to the TP.
- AP_WHEN_SESSION_ALLOCATED specifies that the LU does not return control to the TP until it allocates a session or encounters one of the errors documented in Return Codes in this topic. (If the session limit is zero, the LU returns control immediately.) If a session is not available, the TP waits for one.
- AP_WHEN_SESSION_FREE specifies that the LU allocates a contention-winner or contention-loser session, if one is available or able to be activated, and returns control to the TP. If an error occurs, (as documented in Return Codes in this topic) the call will return immediately with the error in the **primary_rc** and **secondary_rc** fields.
- AP_WHEN_CONWINNER_ALLOC specifies that the LU does not return control until it allocates a contention-winner session or encounters one of the errors documented in Return Codes in this topic. (If the session limit is zero, the LU returns control immediately.) If a session is not available, the TP waits for one.
- AP_WHEN_CONV_GROUP_ALLOC specifies that the LU does not return control to the TP until it allocates the session specified by **conv_group_id** or encounters one of the errors documented in Return Codes in this topic. If a session is not available, the TP waits for it to become free.

Note that AP_IMMEDIATE is the only value for **rtn_ctl** that will never cause a new session to start. For values other than AP_IMMEDIATE, if an appropriate session is not immediately available, Microsoft® Host Integration Server 2000 will try to start one. This will cause the on-demand connection to be activated.

reserv5
A reserved field.
conv_group_id
Supplied/returned parameter. Specifies the identifier of the conversation group from which the session should be allocated. The **conv_group_id** is required only if **rtn_ctl** is set to WHEN_CONV_GROUP_ALLOC. When **rtn_ctl** specifies a different value and the **primary_rc** is AP_OK, this is a returned value.
sense_data
Returned parameter. Indicates an allocation error (retry or no-retry) and contains sense data.
plu_alias
Supplied parameter. Specifies the alias by which the partner LU is known to the local TP.

The **plu_alias** must match the name of a partner LU established during configuration.

The parameter is an 8-byte ASCII character string. It can consist of the following ASCII characters:

- Uppercase letters
- Numerals 0 through 9
- Spaces
- Special characters $, #, %, and @

The first character of this string cannot be a space.

If the value of this parameter is fewer than eight bytes, pad it on the right with ASCII spaces (0x20).

If you want to specify the partner LU with the **fqplu_name** parameter, fill this parameter with binary zeros.

For a user or group using TPs, 5250 emulators, and/or APPC applications, the system administrator can assign default local and remote LUs. In this case, the field is left blank or null and the default LUs are accessed when the user or group member starts an APPC program. For more information on default LUs, see the *Microsoft Host Integration Server 2000 online books*.

mode_name
Supplied parameter. Specifies the name of a set of networking characteristics defined during configuration.

The value of **mode_name** must match the name of a mode associated with the partner LU during configuration.

The parameter is an 8-byte EBCDIC character string. It can consist of characters from the type A EBCDIC character set:

- Uppercase letters
- Numerals 0 through 9
- Special characters $, #, and @

The first character in the string must be an uppercase letter or a special character.

Do not use SNASVCMG in a mapped conversation. SNASVCMG is a reserved **mode_name** used internally by APPC.

tp_name
Supplied parameter. Specifies the name of the invoked TP. The value of **tp_name** specified by **MC_ALLOCATE** in the invoking TP must match the value of **tp_name** specified by RECEIVE_ALLOCATE in the invoked TP.

The parameter is a 64-byte EBCDIC character string and is case-sensitive. The **tp_name** parameter can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If **tp_name** is fewer than 64 bytes, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention is that a service TP name can have up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The other characters are from the type AE EBCDIC character set.

security
Supplied parameter. Provides the information that the partner LU requires to validate access to the invoked TP.

Based on the conversation security established for the invoked TP during configuration, use one of the following values:

- AP_NONE for an invoked TP that uses no conversation security.
- AP_PGM for an invoked TP that uses conversation security and thus requires a user identifier and password. Supply this information through the **user_id** and **pwd** parameters.
- AP_PROXY_PGM for an invoked TP with privileged proxy that uses conversation security and thus requires a user identifier and password. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields.
- AP_PROXY_SAME for a TP that has been invoked using privileged proxy with a valid user identifier and password supplied by the proxy, which in turn invokes another TP. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields.

  For example, assume that TP A invokes TP B with a valid user identifier and password supplied by the privileged proxy, and TP B in turn invokes TP C. If TP B specifies the value AP_PROXY_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator tells TP C to not require the password (if TP C is configured to accept an already-verified indicator).

- AP_PROXY_STRONG for an invoked TP with privileged proxy that uses conversation security and thus requires a user identifier and password provided by the privileged proxy mechanism. Pointers must be set up for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated. The application does not need to set the **user_id** and **pwd** fields. AP_PROXY_STRONG differs from AP_PROXY_PGM in that AP_PROXY_STRONG does not allow clear-text passwords. If the remote system does not support encrypted passwords (strong conversation security), then this call fails.
- AP_SAME for a TP that has been invoked with a valid user identifier and password, which in turn invokes another TP.

  For example, assume that TP A invokes TP B with a valid user identifier and password, and TP B in turn invokes TP C. If TP B specifies the value AP_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator tells TP C to not require the password (if TP C is configured to accept an already-verified indicator).

When AP_SAME is used in an **MC_ALLOCATE** verb, your application must always provide values for the **user_id** and **pwd** parameters in the verb control block. Depending on the properties negotiated between Host Integration Server 2000 and the peer LU, the MC_ALLOCATE verb will send one of 3 kinds of Attach (FMH-5) messages, in this order of precedence:

1. If the LUs have negotiated "already verified" security, then the Attach sent by Host Integration Server 2000 will not include the contents of the pwd parameter field specified in the VCB.
2. If the LUs have negotiated "persistent verification" security, then the Attach sent by Host Integration Server 2000 will include the pwd parameter specified in the VCB, but only when the Attach is the first for the specified user_id parameter since the start of the LU-LU session, and will omit the pwd parameter on all subsequent Attaches (issued by your application or any other application using this LU-LU-mode triplet).
3. If the LUs have not negotiated either of the above, then the Attach sent by Host Integration Server 2000 will omit both the user_id and pwd parameters on all Attaches.

   Your application cannot tell which mode of security has been negotiated between the LUs, nor can it tell whether the **MC_ALLOCATE** verb it is issuing is the first for that LU-LU-mode triplet. So your application must always set the **user_id** and **pwd** parameter fields in the VCB when **security** is set to AP_SAME.

   For more information on persistent verification and already verified security, see the SNA Formats Guide, section "FM Header 5: Attach (LU 6.2)".

- AP_STRONG for an invoked TP that uses conversation security and thus requires a user identifier and password. Supply this information through the **user_id** and **pwd** parameters. AP_STRONG differs from AP_PGM in that AP_STRONG does not allow clear-text passwords. If the remote system does not support encrypted passwords (strong conversation security), then this call fails.

  If the APPC automatic logon feature is to be used, **security** must be set to AP_PGM. See the Remarks section for details.

reserv6
A reserved field.

pwd
Supplied parameter. Specifies the password associated with **user_id**.

The **pwd** parameter is required only if **security** is set to AP_PGM or AP_SAME. It must match the password for **user_id** that was established during configuration.

The **pwd** parameter is a 10-byte EBCDIC character string and is case-sensitive. It can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If the password is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

If the APPC automatic logon feature is to be used, the **pwd** character string must be hard-coded to MS$SAME. See the Remarks section for details.

user_id
Supplied parameter. Specifies the user identifier required to access the partner TP. It is required only if the security parameter is set to AP_PGM or AP_SAME.

The **user_id** parameter is a 10-byte EBCDIC character string and is case-sensitive. It must match one of the user identifiers configured for the partner TP.

The parameter can consist of the following EBCDIC characters:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, @, and period (.)

If **user_id** is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

If the APPC automatic logon feature is to be used, the **user_id** character string must be hard-coded to MS$SAME. See the Remarks section for details.

pip_dlen
Supplied parameter. Specifies the length of the program initialization parameters (PIP) to be passed to the partner TP. The range

is from 0 through 32767.

pip_dptr

Supplied parameter. Specifies the address of the buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

PIP data can consist of initialization parameters or environmental setup information required by a partner TP or remote operating system. The PIP data must follow the general data stream (GDS) format. For more information, see your IBM SNA manual(s).

For Windows NT, Windows 95, Windows 98, and Windows 3.x, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For OS/2, the PIP data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The PIP data buffer must fit entirely on the segment.

reserv7

A reserved field.

fqplu_name

Supplied parameter. Specifies the fully qualified name of the partner LU. This must match the fully qualified name of the local LU defined in the remote node. The parameter consists of two type A EBCDIC character strings for the NETID and the LU name of the partner LU. The names are separated by an EBCDIC period (.).

This name must be provided if no **plu_alias** is specified. It can consist of the following EBCDIC characters:

- Uppercase letters
- Numerals 0 through 9
- Special characters $, #, and @

If the value of this parameter is fewer than 17 bytes, pad it on the right with EBCDIC spaces (0x40).

reserv8

A reserved field.

proxy_user

Supplied parameter. Specifies a LPWSTR pointing to a UNICODE string containing the user name to be impersonated using the privileged proxy feature. This field can only be used when the AP_EXTD_VCB bit is set on the **opext** field, indicating an extended VCB.

proxy_domain

Supplied parameter. Specifies a LPWSTR pointing to a UNICODE string containing the domain name of the user to be impersonated using the privileged proxy feature. This field can only be used when the AP_EXTD_VCB bit is set on the **opext** field, indicating an extended VCB.

reserv9

A reserved field.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL

Primary return code; the supplied parameter **rtn_ctl** specified immediate (AP_IMMEDIATE) return of control to the TP, and the local LU did not have an available contention-winner session.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_RETURN_CONTROL

Secondary return code; the value specified for **rtn_ctl** was invalid.

AP_BAD_SECURITY

Secondary return code; the value specified for **security** was invalid.

AP_BAD_SYNC_LEVEL

Secondary return code; the value specified for **sync_level** was invalid.

AP_BAD_TP_ID

Secondary return code; the value specified for **tp_id** was invalid.

AP_PIP_LEN_INCORRECT

Secondary return code; the value of **pip_dlen** was greater than 32767.

AP_UNKNOWN_PARTNER_MODE

Secondary return code; the value specified for **mode_name** was invalid.

AP_BAD_PARTNER_LU_ALIAS

Secondary return code; APPC did not recognize the supplied **partner_lu_alias**.

AP_NO_USE_OF_SNASVCMG

Secondary return code; SNASVCMG is not a valid value for **mode_name**.

AP_INVALID_DATA_SEGMENT

Secondary return code; the PIP data was longer than the allocated data segment, or the address of the PIP data buffer was wrong.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after **MC_ALLOCATE**.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **MC_ALLOCATE**, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for an Host Integration Server 2000 system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

MC_ALLOCATE establishes a mapped conversation.

The conversation state is RESET when the TP issues this verb. After successful execution (**primary_rc** is AP_OK), the state changes to SEND. If the verb does not execute, the state remains unchanged.

Several parameters of **MC_ALLOCATE** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

To send the **MC_ALLOCATE** request immediately, the invoking TP can issue MC_FLUSH or MC_CONFIRM immediately after **MC_ALLOCATE**. Otherwise, the **MC_ALLOCATE** request accumulates with other data in the local LU's send buffer until the buffer is full.

By issuing **MC_CONFIRM** after **MC_ALLOCATE**, the invoking TP can immediately determine whether the allocation was successful (if **synclevel** is set to AP_CONFIRM_SYNC_LEVEL).

Normally, the value of the **MC_ALLOCATE** verb's **mode_name** parameter must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU.

If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**.

Host Integration Server 2000 supports a feature called password substitution. This is a security feature supported by the latest version of the OS/400 operating system (V3R1) which encrypts any password that flows between two nodes on an Attach message. A password flows on an Attach whenever someone invokes an APPC transaction program specifying a user identifier and password. For example, this happens whenever anyone logs on to an AS/400.

Support for password substitution is indicated by setting bit 5 in byte 23 of the BIND request to 1 (which indicates that password substitution is supported). If the remote system sets this bit in the BIND response, Host Integration Server 2000 automatically encrypts the LU 6.2 conversation security password included in the FMH-5 Attach message. Host Integration Server 2000 APPC applications automatically take advantage of this feature by setting the **security** field of the VCB to AP_PGM or AP_STRONG in the **MC_ALLOCATE** request.

If an APPC application wants to force an encrypted password to flow, the application can specify AP_STRONG for the **security** field in the VCB in the **MC_ALLOCATE** request. This option is implemented as defined in OS/400 V3R1, and is documented in the OS/400 CPIC programmer reference as CM_SECURITY_PROGRAM_STRONG, where the LU 6.2 **pwd** (password) field is encrypted before it flows over the physical network.

The password substitution features is currently only supported by OS/400 V3R1 or later. If the remote system does not support this feature, Host Integration Server 2000 will UNBIND the session with the sense code of 10060006. The two nodes negotiate whether or not they support this feature in the BIND exchange. Host Integration Server 2000 sets a bit in the BIND, and also adds some random data on the BIND for encryption. If the remote node supports password substitution, it sets the same bit in the BIND response, and adds some (different) random data for decryption.

SNA Server version 3.0 with Service Pack 1 or later and SNA Server version 4.0 support automatic logon for APPC applications. This feature requires specific configuration by the network administrator: The APPC application must be invoked on the LAN side from a client of Host Integration Server 2000. The client must be logged into a Windows NT domain, but can be any platform that supports Host Integration Server 2000's APPC APIs.

The client application is coded to use "program" level security, with a special hard-coded APPC user name MS$SAME and password MS$SAME. When this session allocation flows from client to Host Integration Server 2000, the Host Integration Server 2000 server looks up the host account and password corresponding to the Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

Note It is illegal for the remote node to set the bit specifying password substitution and not add the random data.

According to IBM, there are implementations of LU 6.2 password substitution that do not support password substitution but do echo the password substitution bit back to Host Integration Server 2000, without specifying any random data. When they do this, Host Integration Server 2000 will UNBIND the session with the sense code 10060006.This sense code is interpreted as:

- 1006 = Required field or parameter missing.
- 0006 = A required subfield of a control vector was omitted.

Host Integration Server 2000 should also log an Event 17 (APPC session activation failure: BIND negative response sent).

The correct solution is for the failing implementation to be fixed. However, as a short-term workaround, the following Host Integration Server 2000 SNA Service registry setting can be set:

HKEY_LOCAL_MACHINE\\SYSTEM\CurrentControlSet\Services\snaservr\parameters\NOPWDSUB: REG_SZ: YES

When this parameter is specified in the registry, Host Integration Server 2000's password substitution support will be disabled.

Several updates have been made to Host Integration Server 2000 to allow a privileged APPC application to open an APPC conversation using the Single Signon feature on behalf of any defined Windows NT user. This is referred to as the privileged proxy feature. An extension has been added to the APPC **MC_ALLOCATE** verb to invoke this feature.

An APPC application becomes privileged by being started in a Windows NT user account that is a member of a special Windows NT group. When a Host Security Domain is configured, Host Integration Server 2000 Manager will define a second Windows NT group for use with the host security features of Host Integration Server 2000. If the user account under which the actual client is running is a member of this second Windows NT group, the client is privileged to initiate an APPC conversation on behalf of any user account defined in the Host Account Cache.

The following describes how the privileged proxy feature works:

The Host Integration Server 2000 administrator creates a Host Security Domain called APP. Host Integration Server 2000 Manager now creates two Windows NT groups. The first group is called APP and the second is called APP_PROXY for this example. Users that are assigned to the APP group are enabled for single signon. Users assigned to the APP_PROXY group are privileged proxies. The administrator adds the Windows NT user AppcUser to the APP_PROXY group using the Users button on the Host Security Domain property dialog box in Host Integration Server 2000 Manager.

The administrator then sets up an APPC application on the Host Integration Server 2000 server to run as a Windows NT service called APPCAPP and that service has been setup to operate under the AppcUser user account. When APPCAPP runs, it opens an APPC session via an ALLOCATE verb using the extended VCB format and specifies the Windows NT username of the desired user, UserA (for example).

The SNA Service sees the session request coming from a connection that is a member of the Host Security Domain APP. The Client/Server interface tells the SNA Service that the actual client is AppcUser.

The SNA Service checks to see if AppcUser is a member of the APP_PROXY group. Because AppcUser is a member of APP_PROXY, the SNA Service inserts the Username/Password for UserA in the APPC Attach (FMH-5) command and sends it off to the partner TP.

In order to support the privileged proxy feature, the APPC application must implement the following program logic:

The APPC application must determine the Windows NT user ID and domain name that it wishes to impersonate.

The APPC application must set the following parameters before calling the **MC_ALLOCATE** verb:

Enable the use of the extended **MC_ALLOCATE** verb control block structure by setting the AP_EXTD_VCB flag in the **opext** field.

Set **security** to AP_PROXY_SAME, AP_PROXY_PGM or AP_PROXY_STRONG.

Set up the pointers for **proxy_user** and **proxy_domain** to point to UNICODE strings containing the user name and domain name of the user to be impersonated.

The application does not need to set up the **user_id** and **pwd** fields in the **MC_ALLOCATE** VCB.

When the APPC application performs the above steps and issues the **MC_ALLOCATE** verb, the Host Integration Server 2000 server will perform a lookup in the host security domain for the specified Windows NT user and set the user ID and password fields in the FMH-5 Attach message sent to the remote system.

MC_CONFIRM

The **MC_CONFIRM** verb sends the contents of the local LU's send buffer and a confirmation request to the partner TP.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_CONFIRM** verb.

```
struct mc_confirm {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned long   conv_id;
    unsigned char   rts_rcvd;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_CONFIRM.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Returned parameter. Identifies the conversation established between the two TPs.

rts_rcvd
Returned parameter. Indicates whether the partner TP issued MC_REQUEST_TO_SEND, which requests the local TP to change the conversation to RECEIVE state.

To change to RECEIVE state operating on Microsoft® Windows NT®, Windows 95, or Windows 98, the local TP can use MC_PREPARE_TO_RECEIVE, MC_RECEIVE_AND_WAIT, or MC_RECEIVE_AND_POST.

To change to RECEIVE state operating on Windows 3.x, the local TP can use **MC_PREPARE_TO_RECEIVE** or **MC_RECEIVE_AND_WAIT**.

To change to RECEIVE state operating on OS/2, the local TP can use **MC_RECEIVE_AND_POST**.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_CONFIRM_ON_SYNC_LEVEL_NONE

Secondary return code; the local TP attempted to use **MC_CONFIRM** in a conversation with a synchronization level of AP_NONE. The synchronization level, established by MC_ALLOCATE, must be AP_CONFIRM_SYNC_LEVEL.

AP_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

AP_CONFIRM_BAD_STATE

Secondary return code; the conversation was not in SEND state.

AP_CONFIRM_NOT_LL_BDY

Secondary return code; the conversation for the local TP was in SEND state, and the local TP did not finish sending a logical record.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after **MC_ALLOCATE**.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE, AP_CONFIRM_SYNC_LEVEL, or AP_SYNCPT) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer has encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued MC_SEND_ERROR. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

In response to **MC_CONFIRM**, the partner TP normally issues MC_CONFIRMED to confirm that it has received the data without error. (If the partner TP encounters an error, it issues MC_SEND_ERROR or abnormally deallocates the conversation.)

The TP can issue **MC_CONFIRM** only if the conversation's synchronization level, established by MC_ALLOCATE, is AP_CONFIRM_SYNC_LEVEL.

The conversation must be in SEND state when the TP issues this verb. State changes, summarized in the following table, are based on the value of the **primary_rc**.

primary_rc	**New state**
AP_OK	No change
AP_ALLOCATION_ERROR	RESET
AP_COMM_SUBSYSTEM_ABENDED AP_COMM_SUBSYSTEM_NOT_LOADED	RESET RESET
AP_CONV_FAILURE_RETRY AP_CONV_FAILURE_NO_RETRY	RESET RESET
AP_DEALLOC_ABEND AP_DEALLOC_ABEND_PROG AP_DEALLOC_ABEND_SVC AP_DEALLOC_ABEND_TIMER	RESET RESET RESET RESET
AP_PROG_ERROR_PURGING AP_SVC_ERROR_PURGING	RECEIVE RECEIVE

MC_CONFIRM waits for a response from the partner TP. A response is generated by one of the following verbs in the partner TP:

- MC_CONFIRMED
- MC_SEND_ERROR
- MC_DEALLOCATE with dealloc_type set to AP_ABEND
- TP_ENDED

By issuing **MC_CONFIRM** after MC_ALLOCATE, the invoking TP can immediately determine whether the allocation was successful (if **synclevel** is set to AP_CONFIRM_SYNC_LEVEL).

Normally, the value of the **MC_ALLOCATE** verb's **mode_name** parameter must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU.

If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**. For more information, see the *Microsoft Host Integration Server 2000 online books*.

Several parameters of **MC_ALLOCATE** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

To send the **MC_ALLOCATE** request immediately, the invoking TP can issue MC_FLUSH or **MC_CONFIRM** immediately after **MC_ALLOCATE**. Otherwise, the **MC_ALLOCATE** request accumulates with other data in the local LU's send buffer until the buffer is full.

MC_CONFIRMED

The **MC_CONFIRMED** verb responds to a confirmation request from the partner TP. It informs the partner TP that the local TP has not detected an error in the received data. Because the TP issuing the confirmation request waits for a confirmation, **MC_CONFIRMED** synchronizes the processing of the two TPs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_CONFIRMED** verb.

```
struct mc_confirmed {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned long   conv_id;
    unsigned char   rts_rcvd;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_CONFIRMED.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Identifies the conversation established between the two TPs. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_CONFIRMED_BAD_STATE

Secondary return code; the conversation is not in CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation must be in one of the following states when the TP issues this verb:

- CONFIRM
- CONFIRM_SEND
- CONFIRM_DEALLOCATE

The new state is determined by the old state — the state of the conversation when the local TP issued **MC_CONFIRMED**. The old state is indicated by the value of the **what_rcvd** parameter of the preceding receive verb. The following state changes are possible:

Old state	**New state**
CONFIRM	RECEIVE
CONFIRM_SEND	SEND

CONFIRM_DEALLOCATE	RESET

Confirmation Requests

A confirmation request is issued by one of the following verbs in the partner TP:

- MC_CONFIRM
- MC_PREPARE_TO_RECEIVE if **ptr_type** is set to AP_SYNC_LEVEL and the conversation's synchronization level (established by MC_ALLOCATE) is AP_CONFIRM_SYNC_LEVEL
- MC_DEALLOCATE if **dealloc_type** is set to AP_SYNC_LEVEL and the conversation's synchronization level (established by **MC_ALLOCATE**) is AP_CONFIRM_SYNC_LEVEL
- MC_SEND_DATA if **type** is set to AP_SEND_DATA_CONFIRM and the conversation's synchronization level (established by **MC_ALLOCATE**) is AP_CONFIRM_SYNC_LEVEL

A confirmation request is received by the local TP through the **what_rcvd** parameter of one of the following verbs:

- MC_RECEIVE_IMMEDIATE
- MC_RECEIVE_AND_WAIT
- MC_RECEIVE_AND_POST for Windows NT, Windows 95, Windows 98, , and OS/2

MC_CONFIRMED is issued by the local TP only if **what_rcvd** contains one of the following values:

- AP_CONFIRM_WHAT_RECEIVED
- AP_CONFIRM_SEND
- AP_CONFIRM_DEALLOCATE

If the **rtn_status** parameter is set to AP_YES, **what_rcvd** can also contain the following values:

- AP_DATA_COMPLETE_CONFIRM
- AP_DATA_COMPLETE_CONFIRM_SEND
- AP_DATA_COMPLETE_CONFIRM_DEALL

MC_DEALLOCATE

The **MC_DEALLOCATE** verb deallocates a conversation between two TPs.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC function rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_DEALLOCATE** verb.

```
struct mc_deallocate {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    reserv3;
    unsigned char    dealloc_type;
    unsigned char    reserv4[2];
    unsigned char    reserv5[4];
    void             (WINAPI *callback)();
    void             *correlator;
    unsigned char    reserv6[4];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_DEALLOCATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Identifies the conversation established between the two TPs. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

dealloc_type
Supplied parameter. Specifies how to perform the deallocation.

For **MC_DEALLOCATE**, use AP_ABEND to deallocate the conversation abnormally. If the conversation is in SEND state when the local TP issues **MC_DEALLOCATE**, APPC sends the contents of the local LU's send buffer to the partner TP before deallocating the conversation. If the conversation is in RECEIVE or PENDING_POST state, APPC purges any incoming data before deallocating the conversation.

A TP should specify AP_ABEND when it encounters an error preventing the successful completion of a transaction.

AP_FLUSH sends the contents of the local LU's send buffer to the partner TP before deallocating the conversation. This value is allowed only if the conversation is in SEND state.

AP_SYNC_LEVEL uses the conversation's synchronization level (established by MC_ALLOCATE) to determine how to deallocate the conversation. This value is allowed only if the conversation is in SEND state.

If the synchronization level of the conversation is AP_NONE, APPC sends the contents of the local LU's send buffer to the

partner TP before deallocating the conversation.

If the synchronization level is AP_CONFIRM_SYNC_LEVEL, APPC sends the contents of the local LU's send buffer and a confirmation request to the partner TP. Upon receiving confirmation from the partner TP, APPC deallocates the conversation. If, however, the partner TP reports an error, the conversation remains allocated.

callback

Supplied parameter. Only present if the AP_EXTD_VCB bit is set in the **opext** member indicating support for Sync Point. This parameter is the address of a user-supplied callback function. If this field is NULL, no notification will be provided.

The prototype of the callback routine is as follows:

```
void WINAPI callback_proc(
struct appc_hdr *vcb,
unsigned char tp_id[8],
unsigned long conv_id,
unsigned short type,
void *correlator
);
```

The callback procedure can take any name, since the address of the procedure is passed to the APPC DLL. The parameters passed to the function are as follows:

vcb

A pointer to the **MC_DEALLOCATE** verb control block that caused the conversation to be deallocated.

tp_id

The TP identifier of the TP that owned the deallocated conversation.

conv_id

The conversation identifier of the deallocated conversation.

type

The type of the message flow that caused the callback to be invoked. Possible values are:

AP_DATA_FLOW Normal data flow on the session.

AP_UNBIND The session was unbound normally.

AP_FAILURE The session terminated due to an outage.

correlator

This value is the **correlator** specified on the **MC_DEALLOCATE** verb.

correlator

Supplied parameter. Only present if the AP_EXTD_VCB bit is set in the **opext** member indicating support for the Sync Point API. This correlator field allows the TP to specify a value it can use to correlate a call to the callback function with, for example, its own internal data structures. This value is returned to the TP as one of the parameters of the callback routine when it is invoked.

reserv4

A reserved field.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_DEALLOC_BAD_TYPE

Secondary return code; the **dealloc_type** parameter was not set to a valid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_DEALLOC_CONFIRM_BAD_STATE

Secondary return code; the conversation was not in SEND state, and the TP attempted to flush the send buffer and send a confirmation request. This attempt occurred because the value of **dealloc_type** was AP_SYNC_LEVEL and the synchronization level of the conversation was AP_CONFIRM_SYNC_LEVEL.

AP_DEALLOC_FLUSH_BAD_STATE

Secondary return code; the conversation was not in SEND state and the TP attempted to flush the send buffer. This attempt occurred because the value of **dealloc_type** was AP_FLUSH or because the value of **dealloc_type** was AP_SYNC_LEVEL and the synchronization level of the conversation was AP_NONE. In either case, the conversation must be in SEND state.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued MC_SEND_ERROR. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued **MC_DEALLOCATE** with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

Depending on the value of the **dealloc_type** parameter, the conversation can be in one of the states indicated in the following table when the TP issues **MC_DEALLOCATE**.

Dealloc_type	**Allowed state**
AP_FLUSH	SEND
AP_SYNC_LEVEL	SEND
AP_ABEND	Any state except RESET
AP_ABEND_PROG	Any state except RESET
AP_ABEND_SVC	Any state except RESET
AP_ABEND_TIMER	Any state except RESET

State changes, summarized in the following table, are based on the value of the **primary_rc**.

Primary_rc	**New state**
AP_OK	RESET
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET

AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE

Before deallocating the conversation, this verb performs the equivalent of one of the following:

- MC_FLUSH, by sending the contents of the local LU's send buffer to the partner LU (and TP).
- MC_CONFIRM, by sending the contents of the local LU's send buffer and a confirmation request to the partner TP.

After this verb has successfully executed, the conversation identifier is no longer valid.

LU 6.2 Sync Point can use an optimization of the message flows known as implied forget. When the protocol specifies that a FORGET PS header is required, the next data flow on the session implies that a FORGET has been received. In the normal situation, the TP is aware of the next data flow when data is received or sent on one of its Sync Point conversations.

However, it is possible that the last message to flow is caused by the conversation being deallocated. In this case, the TP is unaware when the next data flow on the session occurs. In order to provide the TP with this notification, the **MC_DEALLOCATE** verb is modified to allow the TP to register a callback function which will be called:

- On the first normal flow transmission (request or response) over the session used by the conversation.
- If the session is unbound before any other data flows.
- If the session is terminated abnormally due to a DLC outage.

The **MC_DEALLOCATE** verb also contains a correlator field member that is returned as one of the parameters when the callback function is invoked. The application can use this parameter in any way (for example, as a pointer to a control block within the application).

The TP can use the *type* parameter passed to the callback function to determine whether the message flow indicates an implied forget has been received.

Note that the **MC_DEALLOCATE** verb will probably complete before the callback routine is called. The conversation is considered to be in RESET state and no further verbs can be issued using the conversation identifier. If the application issues a TP_ENDED verb before the next data flow on the session, the callback routine will not be invoked.

Host Integration Server 2000 allows TPs to deallocate conversations immediately after sending data by specifying the **type** parameter on MC_SEND_DATA as AP_SEND_DATA_DEALLOC_*. However, the **MC_SEND_DATA** verbs do not contain the implied forget callback function. TPs wishing to receive implied forget notification must issue **MC_DEALLOCATE** explicitly.

MC_FLUSH

The **MC_FLUSH** verb sends the contents of the local LU's send buffer to the partner LU (and TP). If the send buffer is empty, no action takes place.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_FLUSH** verb.

```
struct mc_flush {
    unsigned short  opcode;
    unsigned char   opext;
    unsigned char   reserv2;
    unsigned short  primary_rc;
    unsigned long   secondary_rc;
    unsigned char   tp_id[8];
    unsigned long   conv_id;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_FLUSH.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_FLUSH_NOT_SEND_STATE

Secondary return code; the conversation was not in SEND state.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.

- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

Data processed by MC_SEND_DATA accumulates in the local LU's send buffer until one of the following happens:

- The local TP issues **MC_FLUSH** (or other verb that flushes the LU's send buffer).
- The buffer is full.

The request generated by MC_ALLOCATE is also buffered.

The conversation must be in SEND state when the TP issues this verb.

There is no state change.

MC_GET_ATTRIBUTES

The **MC_GET_ATTRIBUTES** verb returns the attributes of the conversation.

The following structure describes the verb control block used by the **MC_GET_ATTRIBUTES** verb.

```
struct mc_get_attributes {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    reserv3;
    unsigned char    sync_level;
    unsigned char    mode_name[8];
    unsigned char    net_name[8];
    unsigned char    lu_name[8];
    unsigned char    lu_alias[8];
    unsigned char    plu_alias[8];
    unsigned char    plu_un_name[8];
    unsigned char    reserv4[2];
    unsigned char    fqplu_name[17];
    unsigned char    reserv5;
    unsigned char    user_id[10];
    unsigned long    conv_group_id;
    unsigned char    conv_corr_len;
    unsigned char    conv_corr[8];
    unsigned char    reserv6[13];
NOTE: The following fields are present when the high bit of opext is set (opext & AP_EXTD_VCB
) != 0.
    unsigned char    luw_id[26];
    unsigned char    sess_id[8];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_GET_ATTRIBUTES.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

sync_level
Returned parameter. Specifies the level of synchronization processing for the conversation. This parameter determines whether the TPs can request confirmation of receipt of data and confirm receipt of data.

AP_NONE indicates that confirmation processing will not be used in this conversation.

AP_CONFIRM_SYNC_LEVEL indicates that TPs can use confirmation processing in this conversation.

AP_SYNCPT indicates that TPs can use Sync Point Level 2 confirmation processing in this conversation.

mode_name
Returned parameter. Specifies the name of a set of networking characteristics. It is a type A EBCDIC character string.

net_name
Returned parameter. Specifies the name of the SNA network containing the local LU used by this TP. It is a type A EBCDIC character string.

lu_name
Returned parameter. Provides the name of the local LU.

lu_alias
Returned parameter. Provides the alias by which the local LU is known to the local TP. It is an ASCII character string.

plu_alias
Returned parameter. Provides the alias by which the partner LU is known to the local TP. It is an ASCII character string.

plu_un_name
Returned parameter. Specifies the uninterpreted name of the partner LU — the name of the partner LU as defined to the system services control point (SSCP). It is a type AE EBCDIC character string. This parameter is returned only if the local LU is dependent.

fqplu_name
Returned parameter. Provides the fully qualified name of the partner LU. It is a type A EBCDIC character string. The field contains the network name, an EBCDIC period, and the partner-LU name.

user_id
Returned parameter. Specifies the user identifier sent by the invoking TP through MC_ALLOCATE to access the invoked TP (if applicable). It is a type AE EBCDIC character string. The field contains the user identifier if the following conditions are true:

- The invoked TP requires conversation security.
- **MC_GET_ATTRIBUTES** was issued by the invoked TP.

Otherwise, the field contains spaces.

conv_group_id
Returned parameter. Specifies the conversation group identifier for the session to which the conversation has been allocated. This is also returned on MC_ALLOCATE and RECEIVE_ALLOCATE.

conv_corr_len
Returned parameter. Specifies the length of the conversation correlator identifier that is returned.

conv_corr
Returned parameter. Specifies the conversation correlator identifier (if any) that the source LU assigns to identify the conversation, which is unique for the source/partner LU pair. It is sent by the source LU on the allocation request.

> **Note** The following fields are present when the high bit of **opext** is set (**opext** & AP_EXTD_VCB) != 0.These fields are only present when when using Sync Point Level 2 support.

luw_id
Logical unit-of-work identifier.

sess_id
Session identifier.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

MC_POST_ON_RECEIPT

The **MC_POST_ON_RECEIPT** verb allows the application to register to receive a notification when data or status arrives at the local LU without actually receiving it at the same time. This verb can only be issued while in RECEIVE state and it never causes a change in conversation state. This verb is only supported on Microsoft® Windows NT® and Microsoft® Windows® 95 by Microsoft® SNA Server version 3.0 with Service Pack 1 or later and by SNA Server version 4.0.

When the TP issues this verb, APPC returns control to the TP immediately. When the specified conditions are satisfied, the Win32® event specified by the **sema** parameter is signalled and the verb completes. Then the TP looks at the return code in the verb control block to determine whether or not any data or status notification has arrived at the local LU and issues an MC_RECEIVE_IMMEDIATE or MC_RECEIVE_AND_WAIT verb to actually receive the data or status notification.

The **MC_POST_ON_RECEIPT** verb implements both the **POST_ON_RECEIPT** and **TEST** verbs as described in the IBM Transaction Programmer's manual for LU Type 6.2.

The following structure describes the verb control block used by the **MC_POST_ON_RECEIPT** verb.

```
struct mc_post_on_receipt {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv1;
    unsigned char    primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned short   reserv2;
    unsigned char    reserv3;
    unsigned char    reserv4;
    unsigned short   max_len;
    unsigned short   reserv5;
    unsigned char *  reserv6;
    unsigned char    reserv7[5];
    unsigned long    sema;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_POST_ON_RECEIPT.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv1
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv2
A reserved field.

reserv3
A reserved field.

reserv4
A reserved field.

max_len
Supplied parameter. Specifies the length of data that triggers APPC to post a notification to the TP.

reserv5
A reserved field.
reserv6
A reserved field.
reserv7
A reserved field.
sema
Supplied parameter. Specifies the handle of a Win32 event. The event should have been created by the TP and the TP is responsible for ensuring that it is reset before a call is made and after the verb completes.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_DATA

Secondary return code; data is available for the program to receive.

AP_NOT_DATA

Secondary return code; information other than data is available for the program to receive.

AP_CANCELLED
Primary return code; the verb was canceled.
AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_INVALID_SEMAPHORE_HANDLE

Secondary return code; the **sema** parameter was not set to a valid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.
AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE.
- The partner TP has encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

AP_DEALLOC_NORMAL
Primary return code; the partner TP has deallocated the conversation without requesting confirmation and issued MC_DEALLOCATE with **dealloc_type** set to one of the following:

- AP_CONFIRM_SYNC_LEVEL
- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

AP_PROG_ERROR_NO_TRUNC
Primary return code; the partner TP has issued MC_SEND_ERROR while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued MC_SEND_ERROR. Data sent but not yet received is purged.

AP_PROG_ERROR_TRUNC
Primary return code; the partner TP has issued MC_SEND_ERROR while the conversation was in SEND state. Data was truncated.

AP_SVC_ERROR_NO_TRUNC
Primary return code; the partner TP (or partner LU) issued MC_SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP was not truncated.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued MC_SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE,

PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

AP_SVC_ERROR_TRUNC

Primary return code; the partner TP (or partner LU) issued MC_SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been truncated.

Remarks

While an **MC_POST_ON_RECEIPT** verb is outstanding, the following verbs can be issued on the same conversation:

GET_ATTRIBUTES

GET_TYPE

MC_DEALLOCATE

MC_RECEIVE_AND_WAIT

MC_RECEIVE_IMMEDIATE

MC_REQUEST_TO_SEND

MC_SEND_ERROR

MC_TEST_RTS

TP_ENDED

Issuing any of the following verbs prior to completion of the asynchronous **MC_POST_ON_RECEIPT** verb causes the **MC_POST_ON_RECEIPT** verb to be canceled (the Win32 event is signaled and the primary return code in the verb control block is set to AP_CANCELLED).

MC_DEALLOCATE

MC_RECEIVE_AND_WAIT

MC_RECEIVE_IMMEDIATE

MC_SEND_ERROR

TP_ENDED

MC_PREPARE_TO_RECEIVE

The **MC_PREPARE_TO_RECEIVE** verb changes the state of the conversation for the local TP from SEND to RECEIVE.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_PREPARE_TO_RECEIVE** verb.

```
struct mc_prepare_to_receive {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    primary_rc;
    unsigned short   reserv2;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    ptr_type;
    unsigned char    locks;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_PREPARE_TO_RECEIVE.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

ptr_type
Supplied parameter. Specifies how to perform the state change.

Use AP_FLUSH to send the contents of the local LU's send buffer to the partner LU (and TP) before changing the conversation's state to RECEIVE.

The AP_SYNC_LEVEL value uses the conversation's synchronization level (established by **MC_ALLOCATE**) to determine how to perform the state change.

If the synchronization level of the conversation is AP_NONE, APPC sends the contents of the local LU's send buffer to the partner TP before changing the conversation's state to RECEIVE. If the synchronization level is AP_CONFIRM_SYNC_LEVEL, APPC sends the contents of the local LU's send buffer and a confirmation request to the partner TP. Upon receiving confirmation from the partner TP, APPC changes the conversation's state to RECEIVE. If, however, the partner TP reports an error, the state changes to RECEIVE or RESET. See the Remarks in this topic.

locks
Supplied parameter. Specifies when APPC should return control to the local TP.

Use this parameter only if **ptr_type** is set to AP_SYNC_LEVEL and the synchronization level of the conversation, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. (Otherwise, the parameter is ignored.)

- AP_LONG indicates that APPC returns control to the local TP when the confirmation and subsequent data from the partner TP arrive at the local LU. (This method results in more efficient use of the network but requires a longer time to return control to the local TP.)

- AP_SHORT indicates that APPC returns control to the local TP when the confirmation from the partner TP arrives at the local LU.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_P_TO_R_INVALID_TYPE

Secondary return code; the **ptr_type** parameter was not set to a valid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_P_TO_R_NOT_SEND_STATE

Secondary return code; the conversation was not in SEND state.

AP_P_TO_R_NOT_LL_BDY

Secondary return code; the local TP did not finish sending a logical record.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Microsoft® Windows NT®, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued MC_SEND_ERROR. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

Before changing the conversation state, this verb performs the equivalent of one of the following:

- MC_FLUSH, by sending the contents of the local LU's send buffer to the partner LU (and TP).
- MC_CONFIRM, by sending the contents of the local LU's send buffer and a confirmation request to the partner TP.

After this verb has successfully executed, the local TP can receive data.

The conversation must be in SEND state when the TP issues this verb.

State changes, summarized in the following table, are based on the value of **primary_rc**.

primary_rc	New state
AP_OK	RECEIVE
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

The conversation does not change to SEND state for the partner TP until the partner TP receives one of the following values through the **what_rcvd** parameter of a subsequent receive verb:

- AP_SEND
- AP_CONFIRM_SEND and replies with MC_CONFIRMED
- AP_DATA_COMPLETE_CONFIRM_SEND and replies with **MC_CONFIRMED**
- AP_DATA_CONFIRM_SEND and replies with **MC_CONFIRMED**

The receive verbs are MC_RECEIVE_AND_POST (Windows NT, Windows 95, Windows 98, and OS/2), MC_RECEIVE_IMMEDIATE, and MC_RECEIVE_AND_WAIT.

MC_RECEIVE_AND_POST

The **MC_RECEIVE_AND_POST** verb receives application data and status information asynchronously. This allows the local TP to proceed with processing while data is still arriving at the local LU.

MC_RECEIVE_AND_POST is only supported under the Microsoft® Windows NT®, Microsoft® Windows® 95, and OS/2 operating systems. For similar functionality under the Windows version 3.*x* graphical environment, use MC_RECEIVE_AND_WAIT in conjunction with WinAsyncAPPC. Specifically, while an asynchronous **MC_RECEIVE_AND_POST** is outstanding, the following verbs can be issued on the same conversation:

- GET_TYPE
- MC_GET_ATTRIBUTES
- MC_REQUEST_TO_SEND
- MC_SEND_ERROR
- MC_TEST_RTS
- TP_ENDED

This allows an application to use an asynchronous **MC_RECEIVE_AND_POST** to receive data. While the **MC_RECEIVE_AND_POST** is outstanding, it can still use **MC_SEND_ERROR** and **REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support. For information on how a TP receives data and how to use this verb, see Remarks in this topic.

The following structure describes the verb control block used by the **MC_RECEIVE_AND_POST** verb.

```
struct mc_receive_and_post {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       reserv4;
    unsigned char       rts_rcvd;
    unsigned char       reserv5;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char FAR * sema;
    unsigned char       reserv6;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_RECEIVE_AND_POST.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the

invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd

Returned parameter. Indicates whether data or conversation status was received.

- AP_CONFIRM_DEALLOCATE indicates that the partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL. The conversation's synchronization level, established by MC_ALLOCATE, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues MC_CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP issued MC_PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL. The conversation's synchronization level, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues **MC_CONFIRMED**, and begins to send data.
- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP issued MC_CONFIRM. Upon receiving this value, the local TP normally issues **MC_CONFIRMED**.
- AP_DATA_COMPLETE indicates, for **MC_RECEIVE_AND_POST**, that the local TP has received a complete data record or the last part of a data record. Upon receiving this value, the local TP normally reissues **MC_RECEIVE_AND_POST** or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data. Otherwise, the local TP examines status information.

  If **primary_rc** contains AP_OK and **what_rcvd** contains AP_SEND, AP_CONFIRM_SEND, AP_CONFIRM_DEALLOCATE, or AP_CONFIRM_WHAT_RECEIVED, see the description of the value (in this section) for the next action the local TP normally takes.

  If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to the MC_DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates, for **MC_RECEIVE_AND_POST**, that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record). Upon receiving this value, the local TP normally reissues **MC_RECEIVE_AND_POST** (or issues another receive verb) to receive the next part of the record.
- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, that the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses MC_SEND_DATA to begin sending data.

rtn_status

Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

- AP_NO specifies that indicators should be returned individually on separate invocations of the verb.
- AP_YES specifies that indicators should be returned together, provided both are available. Both can be returned when:

  The receive buffer is large enough to hold all of the data that precedes the status indicator.

  The data is the last data record before the status indicator.

rts_rcvd

Returned parameter. Indicates whether the partner TP issued MC_REQUEST_TO_SEND.

- AP_YES indicates that the partner TP issued **MC_REQUEST_TO_SEND**, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **MC_REQUEST_TO_SEND**.

max_len

Supplied parameter. Specifies the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

The value must not exceed the length of the buffer to contain the received data. The offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

dlen

Returned parameter. Specifies the number of bytes of data received. Data is stored in the buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr

Supplied parameter. Provides the address of the buffer to contain the data received by the local LU.

For the Windows NT, Windows 95, and Windows 98 operating systems, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the **DosAllocSeg** function with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

sema
Supplied parameter. Provides the address of the semaphore that APPC is to clear when the asynchronous receiving operation is finished. On OS/2, the **sema** parameter is either a RAM or system semaphore. On Windows NT, Windows 95, and Windows 98 the **sema** parameter is an event handle obtained by calling either the **CreateEvent** or **OpenEvent** Win32® function.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.

AP_DATA_COMPLETE_CONFIRM_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.

AP_DATA_COMPLETE_CONFIRM

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.

AP_DATA_COMPLETE_CONFIRM_DEALL

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.

AP_DEALLOC_NORMAL
Primary return code; the partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_FLUSH or AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE.

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_INVALID_SEMAPHORE_HANDLE

Secondary return code; the address of the RAM semaphore or system semaphore handle was invalid.

> APPC cannot trap all invalid semaphore handles. If the TP passes a bad RAM semaphore handle, a protection violation results.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_AND_POST_BAD_STATE

Secondary return code; the conversation was not in RECEIVE or SEND state when the TP issued this verb.

AP_RCV_AND_POST_NOT_LL_BDY

Secondary return code; the conversation was in SEND state; the TP began but did not finish sending a logical record.

AP_CANCELED

Primary return code; the local TP issued one of the following verbs, which canceled **MC_RECEIVE_AND_POST**:

MC_DEALLOCATE with **dealloc_type** set to AP_ABEND

MC_SEND_ERROR

TP_ENDED

Issuing one of these verbs causes the semaphore to be cleared.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC
Primary return code; the partner TP issued MC_SEND_ERROR while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **MC_SEND_ERROR**. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received is one data record.

The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it. The receive verbs are **MC_RECEIVE_AND_POST** (Windows NT, Windows 95, Windows 98, , and OS/2), MC_RECEIVE_AND_WAIT, and MC_RECEIVE_IMMEDIATE.

1. The local TP issues the receive verb again. This has one of the following effects:

    - If the partner TP has sent more data, the local TP begins to receive a new unit of data.
    - If the partner TP has finished sending data or is waiting for confirmation, status information (available through **what_rcvd**) indicates the next action the local TP normally takes.

The following procedure shows tasks performed by the local TP in using **MC_RECEIVE_AND_POST**.

To use MC_RECEIVE_AND_POST

1. For the Windows NT, Windows 95, and Windows 98 operating systems, the TP retrieves the WinAsyncAPPC message number by calling the **RegisterWindowMessage** API or allocating a semaphore. The **sema** field should be set to NULL if the application expects to be notified through the Windows message mechanism.

APPC sends the Windows message or clears the semaphore when the local TP finishes receiving data.

For the OS/2 operating system, the TP uses the **DosSemSet** function to set the semaphore pointed to by **sema**.

The semaphore will remain set while the local TP receives data asynchronously. APPC will clear the semaphore when the local TP finishes receiving data.

1. The TP issues MC_RECEIVE_AND_POST.
2. The TP checks the value of **primary_rc**.

If **primary_rc** is AP_OK, the receive buffer (pointed to by **dptr**) is asynchronously receiving data from the partner TP. While receiving data asynchronously, the local TP can:

- Perform tasks not related to this conversation.
- Issue MC_REQUEST_TO_SEND.
- Gather information about this conversation by issuing GET_TYPE, MC_GET_ATTRIBUTES, or MC_TEST_RTS.
- Prematurely cancel **MC_RECEIVE_AND_POST** by issuing MC_DEALLOCATE with **dealloc_type** set to AP_ABEND; MC_SEND_ERROR; or TP_ENDED.

If, however, **primary_rc** is not AP_OK, **MC_RECEIVE_AND_POST** has failed. In this case, the local TP does not perform the next two tasks.

1. For the Windows NT, Windows 95, and Windows 98 operating systems, when the TP finishes receiving data asynchronously, APPC issues the WinAsyncAPPC Windows message or clears the semaphore.

For the OS/2 operating system, the TP uses the **DosSemWait** function to wait for APPC to clear the semaphore pointed to by **sema**. When the TP finishes receiving data asynchronously, APPC clears the semaphore. To prevent the local TP from waiting, have it test the semaphore (invoking **DosSemWait** with **Timeout** set to zero) until APPC clears the semaphore.

1. The TP checks the new value of **primary_rc**.

If **primary_rc** is AP_OK, the local TP can examine the other returned parameters and manipulate the asynchronously received data.

If **primary_rc** is not AP_OK, only **secondary_rc** and **rts_rcvd** (request-to-send received) are meaningful.

Conversation State Effects

The conversation must be in RECEIVE or SEND state when the TP issues this verb.

Issuing **MC_RECEIVE_AND_POST** while the conversation is in SEND state has the following effects:

- The local LU sends the information in its send buffer and a SEND indicator to the partner TP.
- The conversation changes to PENDING_POST state; the local TP is ready to receive information from the partner TP asynchronously.

The conversation changes states twice:

- Upon initial return of the verb, if **primary_rc** contains AP_OK, the conversation changes to PENDING_POST state.
- After completion of the verb, the state changes depending on the value of the following:

The **primary_rc** parameter

The **what_rcvd** parameter if **primary_rc** is AP_OK

The following table shows the new state associated with each value of **what_rcvd** when **primary_rc** is AP_OK.

what_rcvd	**New state**
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	RECEIVE
AP_DATA_COMPLETE	RECEIVE
AP_DATA_INCOMPLETE	RECEIVE
AP_SEND	SEND

AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table shows the new state associated with each value of **primary_rc** other than AP_OK.

primary_rc	**New state**
AP_CANCELED	No change
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_PROG_ERROR_NO_TRUNC	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_NO_TRUNC	RECEIVE
AP_PROG_ERROR_TRUNC	RECEIVE
AP_SVC_ERROR_TRUNC	RECEIVE

MC_RECEIVE_AND_WAIT

The **MC_RECEIVE_AND_WAIT** verb receives any data that is currently available from the partner TP. If no data is currently available, the local TP waits for data to arrive.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call. To allow full use to be made of the asynchronous support, asynchronously issued **MC_RECEIVE_AND_WAIT** verbs have been altered to act like MC_RECEIVE_AND_POST verbs. Specifically, while an asynchronous **MC_RECEIVE_AND_WAIT** is outstanding, the following verbs can be issued on the same conversation:

- GET_TYPE
- MC_GET_ATTRIBUTES
- MC_REQUEST_TO_SEND
- MC_SEND_ERROR
- MC_TEST_RTS
- TP_ENDED

This allows an application, and in particular, a 5250 emulator, to use an asynchronous **MC_RECEIVE_AND_WAIT** to receive data. While the **MC_RECEIVE_AND_WAIT** is outstanding, it can still use **MC_SEND_ERROR** and **MC_REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support.

The following structure describes the verb control block used by the **MC_RECEIVE_AND_WAIT** verb.

```
struct mc_receive_and_wait {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       reserv4;
    unsigned char       rts_rcvd;
    unsigned char       reserv5;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char       reserv6[5];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_RECEIVE_AND_WAIT.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Specifies the conversation identifier.

The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd
Returned parameter. Indicates whether data or conversation status was received.

- AP_CONFIRM_DEALLOCATE indicates that the partner TP has issued MC_DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues MC_CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP has issued MC_PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues **MC_CONFIRMED** and begins to send data.
- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP has issued MC_CONFIRM. Upon receiving this value, the local TP normally issues **MC_CONFIRMED**.
- AP_DATA_COMPLETE indicates, for **MC_RECEIVE_AND_WAIT**, that the local TP has received a complete data record or the last part of a data record. Upon receiving this value, the local TP normally reissues **MC_RECEIVE_AND_WAIT** or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data.

  Otherwise, the local TP examines status information, if **primary_rc** contains AP_OK and **what_rcvd** contains AP_SEND, AP_CONFIRM_SEND, AP_CONFIRM_DEALLOCATE, or AP_CONFIRM_WHAT_RECEIVED.

  See Return Codes in this topic for the next action the local TP normally takes.

  If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to MC_DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record). Upon receiving this value, the local TP normally reissues **MC_RECEIVE_AND_WAIT** (or issues another receive verb) to receive the next part of the record.
- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, that the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses MC_SEND_DATA to begin sending data.

rtn_status
Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

- AP_NO specifies that indicators should be returned individually on separate invocations of the verb.
- AP_YES specifies that indicators should be returned together, provided both are available. Both can be returned when:

  The receive buffer is large enough to hold all of the data that precedes the status indicator.

  The data is the last data record before the status indicator.

rts_rcvd
Returned parameter. Contains the request-to-send indicator.

- AP_YES indicates that the partner TP has issued MC_REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **MC_REQUEST_TO_SEND**.

max_len
Supplied parameter. Indicates the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

For the Windows NT, Windows 95, and Windows 98 operating systems and the Windows graphical environment, this value must not exceed the length of the buffer to contain the received data.

For the OS/2 operating system, the offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

By issuing **MC_RECEIVE_AND_WAIT** with **max_len** set to zero, the local TP can determine whether the partner TP has data to send, seeks confirmation, or has changed the conversation state.

dlen
Returned parameter. Indicates the number of bytes of data received. Data is stored in the buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr

Supplied parameter. Provides the address of the buffer to contain the data received by the local TP.

For the Windows NT, Windows 95, and Windows 98 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the **DosAllocSeg** function with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

For the Windows environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.

AP_DATA_COMPLETE_CONFIRM_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.

AP_DATA_COMPLETE_CONFIRM

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.

AP_DATA_COMPLETE_CONFIRM_DEALL

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.

AP_DEALLOC_NORMAL

Primary return code; the partner TP has deallocated the conversation without requesting confirmation and issued MC_DEALLOCATE with **dealloc_type** set to one of the following:

- AP_CONFIRM_SYNC_LEVEL
- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_AND_WAIT_BAD_STATE

Secondary return code; the conversation was not in RECEIVE or SEND state when the TP issued this verb.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY

Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY

Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single

conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC
Primary return code; the partner TP has issued MC_SEND_ERROR while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **MC_SEND_ERROR**. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received is one data record.

The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it.

The receive verbs are MC_RECEIVE_AND_POST (Windows NT, Windows 95, Windows 98, , and OS/2 operating systems), **MC_RECEIVE_AND_WAIT**, and MC_RECEIVE_IMMEDIATE.

1. The local TP issues the receive verb again. This has one of the following effects:

    - If the partner TP has sent more data, the local TP begins to receive a new unit of data.
    - If the partner TP has finished sending data or is waiting for confirmation, status information (available through the **what_rcvd** parameter) indicates the next action the local TP normally takes.

The conversation must be in RECEIVE or SEND state when the TP issues this verb.

Issuing the Verb in SEND State

Issuing **MC_RECEIVE_AND_WAIT** while the conversation is in SEND state has the following effects:

- The local LU sends the information in its send buffer and a SEND indicator to the partner TP.
- The conversation changes to RECEIVE state; the local TP waits for the partner TP to send data.

State Change

The new conversation state is determined by the following factors:

- The state the conversation is in when the TP issues the verb.
- The **primary_rc** parameter.
- The **what_rcvd** parameter if **primary_rc** contains AP_OK.

Verb Issued in SEND State

The following table details the state changes when **MC_RECEIVE_AND_WAIT** is issued in SEND state and **primary_rc** is AP_OK.

what_rcvd	New state
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	RECEIVE
AP_DATA_COMPLETE	RECEIVE
AP_DATA_INCOMPLETE	RECEIVE
AP_SEND	No change
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when **MC_RECEIVE_AND_WAIT** is issued in SEND state and **primary_rc** is not AP_OK.

primary_rc	New state
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_PROG_ERROR_NO_TRUNC	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_NO_TRUNC	RECEIVE

Verb Issued in RECEIVE State

The following table details the state changes when **MC_RECEIVE_AND_WAIT** is issued in RECEIVE state and **primary_rc** is AP_OK.

what_rcvd	New state
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	No change
AP_DATA_COMPLETE	No change
AP_DATA_INCOMPLETE	No change
AP_SEND	SEND
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when **MC_RECEIVE_AND_WAIT** is issued in RECEIVE state and **primary_rc** is not AP_OK.

primary_rc	New state

AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	No change
AP_PROG_ERROR_NO_TRUNC	No change
AP_SVC_ERROR_PURGING	No change
AP_SVC_ERROR_NO_TRUNC	No change
AP_PROG_ERROR_TRUNC	No change
AP_SVC_ERROR_TRUNC	No change

MC_RECEIVE_IMMEDIATE

The **MC_RECEIVE_IMMEDIATE** verb receives any data currently available from the partner TP. If no data is available, the local TP does not wait. To avoid blocking the conversation, the Microsoft® Windows NT®, Microsoft® Windows® 95, and Windows version 3.*x* systems can issue MC_RECEIVE_AND_WAIT in conjunction with WinAsyncAPPC. For OS/2 systems, use MC_RECEIVE_AND_POST.

The following structure describes the verb control block used by the **MC_RECEIVE_IMMEDIATE** verb.

```
struct mc_receive_immediate {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       reserv4;
    unsigned char       rts_rcvd;
    unsigned char       reserv5;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char       reserv6[5];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_RECEIVE_IMMEDIATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd
Returned parameter. Contains information received with the incoming data:

- AP_CONFIRM_DEALLOCATE indicates that the partner TP has issued MC_DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues MC_CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP has issued MC_PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **MC_ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues **MC_CONFIRMED**, and begins to send data.
- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP has issued MC_CONFIRM. Upon receiving this value, the local TP normally issues **MC_CONFIRMED**.
- AP_DATA_COMPLETE indicates, for **MC_RECEIVE_IMMEDIATE** in mapped conversations, that the local TP has received a complete data record or the last part of a data record. Upon receiving this value, the local TP normally reissues

MC_RECEIVE_IMMEDIATE or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data.

Otherwise, the local TP examines status information if **primary_rc** contains AP_OK and **what_rcvd** contains any of these values:

AP_SEND

AP_CONFIRM_SEND

AP_CONFIRM_DEALLOCATE

AP_CONFIRM_WHAT_RECEIVED

See the description of the value in Return Codes in this topic for the next action the local TP normally takes.

If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to MC_DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates for **MC_RECEIVE_IMMEDIATE** in mapped conversations that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record). Upon receiving this value, the local TP normally reissues **MC_RECEIVE_IMMEDIATE** (or issues another receive verb) to receive the next part of the record.
- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses MC_SEND_DATA to begin sending data.

rtn_status
Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

- AP_NO specifies that indicators should be returned individually on separate invocations of the verb.
- AP_YES specifies that indicators should be returned together, provided both are available. Both can be returned when:

  The receive buffer is large enough to hold all of the data that precedes the status indicator.

  The data is the last data record before the status indicator.

rts_rcvd
Returned parameter. Contains the request-to-send indicator. Possible values are:

- AP_YES indicates that the partner TP has issued MC_REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **MC_REQUEST_TO_SEND**.

max_len
Supplied parameter. Indicates the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

For the Windows NT, Windows 95, and Windows 98 operating systems and the Windows graphical environment, this value must not exceed the length of the buffer to contain the received data.

For the OS/2 operating system, the offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

By issuing **MC_RECEIVE_IMMEDIATE** with **max_len** set to zero, the local TP can determine whether the partner TP has data to send, seeks confirmation, or has changed the conversation state.

dlen
Returned parameter. Provides the number of bytes of data received. Data is stored in a buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr
Supplied parameter. Address of the buffer to contain the data received by the local TP.

For the Windows NT, Windows 95, and Windows 98 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.

AP_DATA_COMPLETE_CONFIRM_SEND

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.

AP_DATA_COMPLETE_CONFIRM

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.

AP_DATA_COMPLETE_CONFIRM_DEALL

Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.

AP_UNSUCCESSFUL
Primary return code; no data is immediately available from the partner TP.

AP_DEALLOC_NORMAL
Primary return code; the partner TP has deallocated the conversation without requesting confirmation. The partner TP issued MC_DEALLOCATE with **dealloc_type** set to one of the following:

- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_IMMD_BAD_STATE

Secondary return code; the conversation was not in RECEIVE state.

AP_ALLOCATION_ERROR

Secondary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_FAILURE_NO_RETRY

Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY

Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC

Primary return code; the partner TP has issued MC_SEND_ERROR while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING

Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, Windows 98, , and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **MC_SEND_ERROR**. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY

Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND

Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received is one data record.

The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it.

The receive verbs are MC_RECEIVE_AND_POST (Windows NT, Windows 95, Windows 98, , and OS/2 operating systems), MC_RECEIVE_AND_WAIT, and **MC_RECEIVE_IMMEDIATE**.

1. The local TP issues the receive verb again. This has one of the following effects:

   - If the partner TP has sent more data, the local TP begins to receive a new unit of data.
   - If the partner TP has finished sending data or is waiting for confirmation, status information (available through **what_rcvd**) indicates the next action the local TP normally takes.

The conversation must be in RECEIVE state when the TP issues this verb.

The new state is determined by **primary_rc**. If **primary_rc** is AP_OK, the new state is determined by **what_rcvd**.

The following table details the state changes when the **primary_rc** is AP_OK.

what_rcvd	**New state**
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	No change
AP_DATA_COMPLETE	No change
AP_DATA_INCOMPLETE	No change
AP_SEND	SEND
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when the **primary_rc** is not AP_OK.

primary_rc	**New state**
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET

AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	No change
AP_PROG_ERROR_NO_TRUNC	No change
AP_SVC_ERROR_PURGING	No change
AP_SVC_ERROR_NO_TRUNC	No change
AP_PROG_ERROR_TRUNC	No change
AP_SVC_ERROR_TRUNC	No change
AP_UNSUCCESSFUL	No change

MC_RECEIVE_LOG_DATA

The **MC_RECEIVE_LOG_DATA** verb allows the user to register to receive the log data associated with an inbound Function Management Header 7 (FMH7) error report. The verb passes a buffer to APPC, and any log data received is placed in that buffer. APPC continues to use this buffer as successive FMH7s arrive until it is provided with another buffer (that is, until the TP issues another **MC_RECEIVE_LOG_DATA** specifying a different buffer or no buffer at all). This verb is only supported on Microsoft® Windows NT® and Microsoft® Windows® 95 by Microsoft® SNA Server version 3.0 with Service Pack 1 or later and by SNA Server version 4.0.

Note that the TP itself is responsible for allocating and freeing the buffer. After the buffer has been passed to APPC, the TP should either issue another **MC_RECEIVE_LOG_DATA** specifying a new buffer or a zero-length buffer, or wait until the conversation has finished before freeing the original buffer.

When an FMH7 is received, APPC copies any associated error log general data stream (GDS) into the buffer. If there is no associated error log variable, the buffer is zeroed out. It is up to the TP to check the buffer whenever a return code from a receive verb indicates that an error has been received.

The following structure describes the verb control block used by the **MC_RECEIVE_LOG_DATA** verb.

```
struct mc_receive_log_data {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv1;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      log_dlen;
    unsigned char FAR * log_dptr;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_RECEIVE_LOG_DATA.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

‹reserv1
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

log_dlen
Supplied parameter. Specifies the maximum length of log data that APPC can place in the buffer (that is, the buffer size). The range is from 0 through 65535. Note that a length of zero here indicates that any previous **MC_RECEIVE_LOG_DATA** verb should be cancelled.

log_dptr
Supplied parameter. Specifies the address of the buffer that APPC will use to store the log data.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

MC_REQUEST_TO_SEND

The **MC_REQUEST_TO_SEND** verb notifies the partner TP that the local TP wants to send data.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_REQUEST_TO_SEND** verb.

```
struct mc_request_to_send {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_REQUEST_TO_SEND.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_R_T_S_BAD_STATE

Secondary return code; the conversation is not in an allowed state when the TP issued this verb.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any of the following states when the TP issues this verb:

CONFIRM

PENDING_POST (OS/2)

RECEIVE

There is no state change.

The request-to-send notification is received by the partner program through the **rts_rcvd** parameter of the following verbs:

- MC_CONFIRM
- MC_RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems)
- MC_RECEIVE_AND_WAIT
- MC_RECEIVE_IMMEDIATE
- MC_SEND_DATA

- MC_SEND_ERROR

It is also indicated by a **primary_rc** of AP_OK on MC_TEST_RTS.

Request-to-send notification is sent to the partner TP immediately; APPC does not wait until the send buffer fills up or is flushed. Consequently, the request-to-send notification may arrive out of sequence. For example, if the local TP is in SEND state and issues MC_PREPARE_TO_RECEIVE followed by **MC_REQUEST_TO_SEND**, the partner TP, in RECEIVE state, may receive the request-to-send notification before it receives the send notification. For this reason, request-to-send can be reported to a TP through a receive verb.

In response to this request, the partner TP can change the conversation to:

- RECEIVE state by issuing **MC_PREPARE_TO_RECEIVE** or MC_RECEIVE_AND_WAIT.
- PENDING_POST state by issuing MC_RECEIVE_AND_POST.

The partner TP can also ignore the request-to-send.

The conversation state changes to SEND for the local TP when the local TP receives one of the following values through the **what_rcvd** parameter of a subsequent receive verb:

- AP_CONFIRM_SEND and replies with MC_CONFIRMED
- AP_DATA_COMPLETE_CONFIRM_SEND and replies with **MC_CONFIRMED**
- AP_SEND

The receive verbs are MC_RECEIVE_AND_POST (Microsoft® Windows NT®, Windows 95, and OS/2 operating systems), MC_RECEIVE_IMMEDIATE, and MC_RECEIVE_AND_WAIT.

MC_SEND_CONVERSATION

The **MC_SEND_CONVERSATION** verb allocates a session between the local LU and partner LU, sends data on the session, and then deallocates the session.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_SEND_CONVERSATION** verb.

```
struct mc_send_conversation {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3[8];
    unsigned char       rtn_ctl;
    unsigned char       reserv4;
    unsigned long       conv_group_id;
    unsigned long       sense_data;
    unsigned char       plu_alias[8];
    unsigned char       mode_name[8];
    unsigned char       tp_name[64];
    unsigned char       security;
    unsigned char       reserv6[11];
    unsigned char       pwd[10];
    unsigned char       user_id[10];
    unsigned short      pip_dlen;
    unsigned char FAR * pip_dptr;
    unsigned char       reserv6;
    unsigned char       fqplu_name[17];
    unsigned char       reserv7[8];
    unsigned short      dlen;
    unsigned char FAR * dptr;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_SEND_CONVERSATION.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

rtn_ctl
Supplied parameter. Specifies how APPC should select a session to allocate for the conversation and when the local LU should return control to the local TP. The allowed values are:

- AP_IMMEDIATE specifies that the LU allocates a contention-winner session, if one is immediately available, and returns

control to the TP.

- AP_WHEN_SESSION_ALLOCATED specifies that the LU does not return control to the TP until it allocates a session or encounters one of the errors described in Return Codes in this topic. If the session limit is zero, the LU returns control immediately. Note that if a session is not available, the TP waits for one.
- AP_WHEN_SESSION_FREE specifies that the LU allocates a contention-winner or contention-loser session, if one is available or able to be activated, and returns control to the TP. If an error occurs (as described in Return Codes in this topic) the call will return immediately with the error in the **primary_rc** and **secondary_rc** fields.
- AP_WHEN_CONWINNER_ALLOC specifies that the LU does not return control until it allocates a contention-winner session or encounters one of the errors described in Return Codes in this topic. If the session limit is zero, the LU returns control immediately. Note that if a session is not available, the TP waits for one.
- AP_WHEN_CONV_GROUP_ALLOC specifies that the LU does not return control to the TP until it allocates the session specified by **conv_group_id** or encounters one of the errors described in Return Codes in this topic. If the session is not available, the TP waits for it to become free.

conv_group_id

Supplied/returned parameter. Used as a supplied parameter when **rtn_ctl** is WHEN_CONV_GROUP_ALLOC to specify the identity of the conversation group from which the session should be allocated. When **rtn_ctl** specifies a different value, and the **primary_rc** is AP_OK, this is a returned value. The purpose of this parameter is to provide a TP with the assurance that the same session will be reallocated and therefore the conversations conducted over the session will occur in the same sequence that they were initiated.

sense_data

Returned parameter. If the primary and secondary return codes indicate an allocation error (retry or no-retry), an SNA-defined sense code is returned.

plu_alias

Supplied parameter. Specifies the alias by which the partner LU is known to the local TP. This parameter must match the name of a partner LU established during configuration. The parameter is an 8-byte, type G ASCII character set that includes:

- Uppercase letters
- Numerals 0 to 9
- Spaces
- Special characters $, #, %, and @

If the value of this parameter is fewer than eight bytes, pad it on the right with ASCII spaces (0x20).

mode_name

Supplied parameter. Specifies the name of a set of networking characteristics defined during configuration. This parameter must match the name of a mode associated with the partner LU during configuration.

The parameter is an 8-byte EBCDIC character string. It can consist of characters from the type A EBCDIC character set, including all EBCDIC spaces. These characters are:

- Uppercase letters
- Numerals 0 to 9
- Special characters $, #, and @

The first character in the string must be an uppercase letter or special character.

In a mapped conversation, the name cannot be SNASVCMG (a reserved mode name used internally by APPC).

tp_name

Supplied parameter. Specifies the name of the invoked TP. The value of **tp_name** specified by MC_ALLOCATE in the invoking TP must match the value of **tp_name** specified by RECEIVE_ALLOCATE in the invoked TP.

The parameter is a 64-byte, case-sensitive, EBCDIC character string. This parameter can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the TP name is fewer than 64 bytes, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention is that a service TP name can have up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The other characters are from the EBCDIC AE character set.

security
Supplied parameter. Specifies the information the partner LU requires in order to validate access to the invoked TP.

- AP_NONE specifies that the invoked TP uses no conversation security.
- AP_PGM specifies that the invoked TP uses conversation security and requires a user identifier and password. Use **user_id** and **pwd** to supply this information.
- AP_SAME specifies that the invoked TP, invoked with a valid user identifier and password, in turn invokes another TP.

For example, assume that TP A invokes TP B with a valid user identifier and password, and TP B in turn invokes TP C. If TP B specifies the value AP_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator indicates to TP C not to require the password (if TP C is configured to accept an already-verified indicator).

pwd
Supplied parameter. Specifies the password associated with **user_id**. This parameter is required only if the security parameter is set to AP_PGM and must match the password for **user_id** that was established during configuration.

This parameter is a 10-byte, case-sensitive, EBCDIC character string. It can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the password is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

user_id
Supplied parameter. Specifies the user identifier required to access the partner TP. This parameter is required only if the security parameter is set to AP_PGM and must match one of the user identifiers configured for the partner TP.

The parameter can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the user identifier is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

pip_dlen
Supplied parameter. Specifies the length of the PIP to be passed to the partner TP. The range for this parameter is from 0 through 32767.

pip_dptr
Supplied parameter. Specifies the address of the buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

PIP data can consist of initialization parameters or environmental setup information required by a partner TP or remote operating system. The PIP data must follow the GDS format. For more information, see your IBM SNA manual(s).

For the Microsoft® Windows NT® and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area.

For the OS/2 operating system, use a shared, unnamed segment for the data buffer. To allocate the segment, issue the function call **DosAllocSeg** with the shared indicator equal to 1. The data buffer must be entirely within the data segment.

fqplu_name
Supplied parameter. Specifies the fully qualified name of the local LU. This parameter must match the fully qualified name of the local LU defined in the remote node. The parameter is made up of two type A EBCDIC character strings (each of up to eight characters), which are the network name (NETID) and the LU name of the partner LU. The names are separated by an EBCDIC period (.). The NETID can be omitted, and if this is the case, the period should also be omitted.

This name must be provided if no **plu_alias** is provided.

Type A EBCDIC characters contain:

- Uppercase letters
- Numerals 0 to 9
- Special characters $, #, and @

If the value of this parameter is fewer than 17 bytes, pad it on the right with EBCDIC spaces (0x40).

dlen

Supplied parameter. Specifies the number of bytes of data to be put in the local LU's send buffer. The range for this parameter is from 0 through 65535.

dptr

Supplied parameter. Specifies the address of the buffer containing the data to be put in the local LU's send buffer.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, use a shared, unnamed segment for the data buffer. To allocate the segment, issue the function call **DosAllocSeg** with the shared indicator equal to 1. The data buffer must be entirely within the data segment.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL

Primary return code; the supplied parameter **rtn_ctl** specified immediate return of the control to the TP (AP_IMMEDIATE), and the local LU did not have an available contention-winner session.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_RETURN_CONTROL

Secondary return code; the value specified for **rtn_ctl** was invalid.

AP_BAD_SECURITY

Secondary return code; the value specified for **security** was invalid.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_PIP_LEN_INCORRECT

Secondary return code; the value of **pip_dlen** was greater than 32767.

AP_UNKNOWN_PARTNER_MODE

Secondary return code; the value specified for **mode_name** was invalid.

AP_BAD_PARTNER_LU_ALIAS

Secondary return code, APPC did not recognize the supplied **partner_lu_alias**.

AP_NO_USE_OF_SNASVCMG

Secondary return code; SNASVCMG is not a valid value for **mode_name**.

AP_INVALID_DATA_SEGMENT

Secondary return code; the PIP data or application data was longer than the allocated data segment, or the address of a data buffer was wrong.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

This verb is issued by the invoking TP to conduct an entire conversation with the remote TP. If the remote TP rejects either the conversation initiation or the data, the invoking TP will not receive notification of the rejection.

The conversation state is RESET when the TP issues this verb. There is no state change.

Several parameters of **MC_SEND_CONVERSATION** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

Normally, the value of **mode_name** must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU. If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**.

MC_SEND_DATA

The **MC_SEND_DATA** verb places data in the local LU's send buffer for transmission to the partner TP.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_SEND_DATA** verb.

```
struct mc_send_data {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       rts_rcvd;
    unsigned char       data_type;
    unsigned short int  dlen;
    unsigned char FAR * dptr ;
    unsigned char       type;
    unsigned char       reserv4;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_SEND_DATA.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter. Provides the request-to-send-received indicator.

- AP_YES indicates that the partner TP has issued MC_REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state. To change to RECEIVE state, the local TP can use MC_PREPARE_TO_RECEIVE, MC_RECEIVE_AND_WAIT, or MC_RECEIVE_AND_POST (Microsoft® Windows NT®, Windows 95, and OS/2 operating systems).
- AP_NO indicates that the partner TP has not issued **MC_REQUEST_TO_SEND**.

data_type
Supplied parameter. Specifies the type of data to be sent if Sync Point is supported. Valid parameters are:

AP_APPLICATION

AP_USER_CONTROL_DATA

AP_PS_HEADER

dlen
Supplied parameter. Specifies the number of bytes of data to be put in the local LU's send buffer. The range is from 0 through 65535.

dptr
Supplied parameter. Specifies the address of the buffer containing the data to be put in the local LU's send buffer.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For OS/2, the data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

type
Supplied parameter. Allows a TP to send data and perform other functions within one API call. For example, you can combine **MC_SEND_DATA** with **type** set to CONFIRM to accomplish the same objective as issuing **MC_SEND_DATA** followed by MC_CONFIRM.

- AP_SEND_DATA_CONFIRM corresponds to **MC_SEND_DATA** followed by **MC_CONFIRM**.
- AP_SEND_DATA_FLUSH corresponds to **MC_SEND_DATA** followed by MC_FLUSH.
- AP_SEND_DATA_DEALLOC_ABEND corresponds to **MC_SEND_DATA** followed by MC_DEALLOCATE with a **dealloc_type** of AP_ABEND.
- AP_SEND_DATA_DEALLOC_FLUSH corresponds to **MC_SEND_DATA** followed by **MC_DEALLOCATE** with a **dealloc_type** of AP_FLUSH.
- AP_SEND_DATA_DEALLOC_SYNC_LEVEL corresponds to **MC_SEND_DATA** followed by **MC_DEALLOCATE** with a **dealloc_type** of AP_SYNC_LEVEL.
- AP_SEND_DATA_P_TO_R_FLUSH corresponds to **MC_SEND_DATA** followed by MC_PREPARE_TO_RECEIVE with a **ptr_type** of AP_FLUSH.
- AP_SEND_DATA_P_TO_R_SYNC_LEVEL corresponds to **MC_SEND_DATA** followed by **MC_PREPARE_TO_RECEIVE** with a **ptr_type** of AP_SYNC_LEVEL and **locks** set to AP_SHORT.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_SEND_DATA_INVALID_TYPE

Secondary return code; the specified type was not recognized by APPC.

AP_SEND_DATA_CONFIRM_SYNC_NONE

Secondary return code; the **type** CONFIRM is not permitted for a conversation that was allocated with a **sync_level** of NONE.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_SEND_DATA_NOT_SEND_STATE

Secondary return code; the local TP issued **MC_SEND_DATA**, but the conversation was not in SEND state.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **MC_SEND_ERROR**. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

Remarks

The conversation must be in SEND state when the TP issues this verb. State changes, based on **primary_rc**, are summarized in the following table.

primary_rc	**New state**
AP_OK	No change
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

MC_SEND_DATA may wait indefinitely because the partner TP has not issued a receive verb. If this occurs, the send buffer may

fill up.

The data collected in the local LU's send buffer is transmitted to the partner LU (and partner TP) when one of the following occurs:

- The send buffer fills up.
- The local TP issues MC_FLUSH, MC_CONFIRM, or MC_DEALLOCATE (or other verb that flushes the LU's send buffer).

MC_SEND_ERROR

The **MC_SEND_ERROR** verb notifies the partner TP that the local TP has encountered an application-level error.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **MC_SEND_ERROR** verb.

```
struct mc_send_error {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       rts_rcvd;
    unsigned char       err_type;
    unsigned char       err_dir;
    unsigned char       reserv4;
    unsigned char       reserv5[2];
    unsigned char       reserv6[4];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_SEND_ERROR.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter. Indicates whether the partner TP issued MC_REQUEST_TO_SEND. Possible values include:

- AP_YES indicates that the partner TP has issued **MC_REQUEST_TO_SEND**, which requests that the local TP change the conversation to RECEIVE state. To change to RECEIVE state, the local TP can use MC_PREPARE_TO_RECEIVE, MC_RECEIVE_AND_WAIT, or MC_RECEIVE_AND_POST.
- AP_NO indicates that the partner TP has not issued **MC_REQUEST_TO_SEND**.

err_type
For a mapped conversation, this parameter is supplied if Sync Point is supported. Valid values are:

AP_PROG

AP_BACKOUT_NO_RESYNC

AP_BACKOUT_RESYNC

err_dir

Supplied parameter. Indicates whether the error is with data just received or with data that is about to be sent. Use this parameter only when the conversation is in SEND_PENDING state. The parameter is ignored otherwise. The following are allowed values:

- AP_RCV_DIR_ERROR indicates that the TP issued **MC_SEND_ERROR** after detecting an error associated with the data just received.
- AP_SEND_DIR_ERROR indicates that the TP issued **MC_SEND_ERROR** after detecting an error associated with data it was going to send. For example, the TP encountered an error while reading data from the disk drive.

reserv3
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_ERROR_DIRECTION

Secondary return code; the specified **err_dir** was not recognized by APPC.

AP_SEND_ERROR_BAD_TYPE

Secondary return code; the value of **err_type** was invalid.

AP_SEND_ERROR_LOG_LL_WRONG

Secondary return code; the LL field of the error log GDS variable did not match the actual length of the data.

The following return codes can be generated when **MC_SEND_ERROR** is issued in any allowed state:

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This may occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

The following return codes can be generated only if **MC_SEND_ERROR** is issued in SEND state:

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after MC_ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **MC_SEND_ERROR**. Data sent but not yet received is purged.

AP_DEALLOC_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

The following return code can be generated only if **MC_SEND_ERROR** is issued in RECEIVE state:

AP_DEALLOC_NORMAL
Primary return code; this return code does not indicate an error.

The partner TP issued MC_DEALLOCATE with **dealloc_type** set to one of the following:

- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

Remarks

The conversation can be in any state except RESET when the TP issues this verb. The conversation state must be SEND_PENDING if **err_dir** is used.

The local TP sends the error notification immediately to the partner TP; it does not hold the information in the local LU's send buffer.

Upon successful execution of this verb, the conversation is in SEND state for the local TP and in RECEIVE state for the partner TP.

The new state is determined by **primary_rc**. Possible state changes are summarized in the following table.

primary_rc	New state
AP_OK	SEND
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

If the conversation is in RECEIVE state when the TP issues **MC_SEND_ERROR**, incoming data is purged by APPC. This data includes:

- Data sent by MC_SEND_DATA.
- Return code indicators.
- Confirmation requests.
- Deallocation requests.

APPC does not purge an incoming request-to-send indicator. APPC replaces purged incoming return code indicators with other return codes. The primary return code AP_OK replaces the following purged return code indicators:

AP_PROG_ERROR_NO_TRUNC

AP_PROG_ERROR_PURGING

AP_PROG_ERROR_TRUNC

AP_SVC_ERROR_NO_TRUNC

AP_SVC_ERROR_PURGING

AP_SVC_ERROR_TRUNC

The primary return code AP_DEALLOC_NORMAL replaces the following purged return code indicators:

AP_ALLOCATION_ERROR

AP_ALLOCATION_FAILURE_NO_RETRY

AP_ALLOCATION_FAILURE_RETRY

AP_CONVERSATION_TYPE_MISMATCH

AP_DEALLOC_ABEND

AP_DEALLOC_ABEND_PROG

AP_DEALLOC_ABEND_SVC

AP_DEALLOC_ABEND_TIMER

AP_PIP_NOT_ALLOWED

AP_PIP_NOT_SPECIFIED_CORRECTLY

AP_SECURITY_NOT_VALID

AP_SYNC_LEVEL_NOT_SUPPORTED

AP_TP_NAME_NOT_RECOGNIZED

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

AP_TRANS_PGM_NOT_AVAIL_RETRY

When the conversation is in SEND_PENDING state, APPC reports the following return codes to the partner TP based on the value in **err_dir**:

AP_PROG_ERROR_PURGING
  The local TP issued **MC_SEND_ERROR** with RECEIVE as the **err_dir**.
AP_PROG_ERROR_NO_TRUNC
  The local TP issued **MC_SEND_ERROR** with SEND as the **err_dir**.

MC_TEST_RTS

The **MC_TEST_RTS** verb determines whether a request-to-send notification has been received from the partner TP.

The following structure describes the verb control block used by the **MC_TEST_RTS** verb.

```
struct mc_test_rts {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_TEST_RTS.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter was returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL
Primary return code; request-to-send notification has not been received.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID
Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

MC_TEST_RTS_AND_POST

The **MC_TEST_RTS_AND_POST** verb allows an application, typically a 5250 emulator, to request asynchronous notification when a partner TP requests send direction. It is not supported on Microsoft® MS-DOS® platforms.

The following structure describes the verb control block used by the **MC_TEST_RTS_AND_POST** verb.

```
struct mc_test_rts_and_post {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3;
    unsigned long       handle;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_M_TEST_RTS_AND_POST.

opext
Supplied parameter. Specifies the verb operation extension, AP_MAPPED_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter was returned by MC_ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

handle
Supplied parameter. On Microsoft® Windows NT® and Microsoft® Windows® 95 this field provides the event handle to set. On Windows 3.x, this field provides the Windows handle to receive the completion message. On OS/2, this field provides the address of the semaphore APPC is to clear when the asynchronous operation is finished.

Return Codes from Initial Verb

AP_OK
Primary return code; the verb executed successfully. Note particularly that a return code of AP_OK from the initial verb does not indicate that **MC_REQUEST_TO_SEND** verb received from the partner TP. It simply indicates that the facility to receive asynchronous notification has been registered.

AP_UNSUCCESSFUL
Primary return code; request-to-send notification has not been received.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with MC_ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **MC_ALLOCATE** request.

When **MC_ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Return Codes from Asynchronous Completion

AP_OK
Primary return code; the request-to-send notification has been received from the partner TP.
AP_CANCELLED
The outstanding **TEST_RTS_AND_POST** verb has been terminated. This will occur if the underlying conversation has been deallocated or an AP_TP_ENDED has been issued.
Note that as with **RECEIVE_AND_POST**, the TP is still responsible for correctly terminating the conversation and possibly terminating the TP. Issuing another verb, such as **RECEIVE_IMMEDIATE,** at this point will indicate the reason for the conversation failure.

Remarks

The conversation can be in any state except RESET when the TP issues this verb. There is no state change.

A common feature of many APPC applications, such as 5250 emulators, is a requirement to detect a partner's request to send.

Currently, this can be done by polling the APPC interface to detect the partner's request. For example, an application can occasionally issue one of the following verbs:

- **MC_TEST_RTS**
- **MC_RECEIVE_IMMEDIATE** and check the **rts_rcvd** field
- **MC_SEND_DATA** of zero bytes, again checking the **rts_rcvd** field.

Some of the problems associated with this polling approach are:

- The application must continually interrupt its main work to poll APPC.
- The partner's request is not detected as soon as it becomes available.
- These approaches are processor-intensive.

The **MC_TEST_RTS_AND_POST** verb allows an application running on Windows NT, Windows 95, Windows 3.*x*, or OS/2, typically a 5250 emulator, to request asynchronous notification when the partner TP requests send direction.

An APPC application typically issues the **MC_TEST_RTS_AND_POST** verb while in SEND state and then continues with its main processing. A request for send direction from the partner TP is indicated asynchronously to the application. After dealing with the partner's request, the application typically returns to SEND state, reissues **MC_TEST_RTS_AND_POST**, and continues.

The **MC_TEST_RTS_AND_POST** verb completes synchronously and the return code AP_OK indicates that a request for asynchronous notification has been registered. It is important to emphasize that this does not indicate that request-to-send was received from the partner TP.

When the partner's request to send is received, the asynchronous event completion occurs. It is important to note that this may be before the completion of the local TP's original **MC_TEST_RTS_AND_POST** verb. This will be the case if the partner's request to send was received before the local TP's **MC_TEST_RTS_AND_POST** verb was issued, or while the local TP's **MC_TEST_RTS_AND_POST** verb was being processed.

POST_ON_RECEIPT

The **POST_ON_RECEIPT** verb allows the application to register to receive a notification when data or status arrives at the local LU without actually receiving it at the same time. This verb can only be issued while in RECEIVE state and it never causes a change in conversation state. This verb is only supported on Microsoft® Windows NT® and Microsoft® Windows® 95 by Microsoft® SNA Server version 3.0 with Service Pack 1 or later and by SNA Server version 4.0.

When the TP issues this verb, APPC returns control to the TP immediately. When the specified conditions are satisfied the Win32® event specified by the **sema** parameter is signalled and the verb completes. Then the TP looks at the return code in the verb control block to determine whether or not any data or status notification has arrived at the local LU and issues a RECEIVE_IMMEDIATE or RECEIVE_AND_WAIT verb to actually receive the data or status notification.

The **POST_ON_RECEIPT** verb implements both the **POST_ON_RECEIPT** and **TEST** verbs as described in the IBM Transaction Programmer's manual for LU Type 6.2.

The following structure describes the verb control block used by the **POST_ON_RECEIPT** verb.

```
struct post_on_receipt {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv1;
    unsigned char    primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned short   reserv2;
    unsigned char    fill;
    unsigned char    reserv4;
    unsigned short   max_len;
    unsigned short   reserv5;
    unsigned char *  reserv6;
    unsigned char    reserv7[5];
    unsigned long    sema;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_POST_ON_RECEIPT.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv1
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv2
A reserved field.

fill
Supplied parameter. Specifies how the local TP receives data. The following values are allowed:

AP_BUFFER

Specifies that APPC should post a notification when the number of data bytes specified by **max_len** have arrived at the local LU, the end of the data has been reached, or information other than data is received.(such as a conversation status, a confirmation,

or a syncpoint request).

AP_LL

Specifies that APPC should post a notification when a complete or truncated logical record is received, when a portion of a logical record is received which is at least equal in length to the length specified by **max_len**, or when information other than data is received.

reserv4
A reserved field.
max_len
Supplied parameter. Specifies the length of data that triggers APPC to post a notification to the TP.
reserv5
A reserved field.
reserv6
A reserved field.
reserv7
A reserved field.
sema
Supplied parameter. Specifies the handle of a Win32 event. The event should have been created by the TP and the TP is responsible for ensuring that it is reset before a call is made and after the verb completes.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_DATA

Secondary return code; data is available for the program to receive.

AP_NOT_DATA

Secondary return code; information other than data is available for the program to receive.

AP_CANCELLED
Primary return code; the verb was canceled.
AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_INVALID_SEMAPHORE_HANDLE

Secondary return code; the **sema** parameter was not set to a valid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.
AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_DEALLOC_NORMAL
Primary return code; the partner TP has deallocated the conversation without requesting confirmation and issued DEALLOCATE with **dealloc_type** set to one of the following:

- AP_CONFIRM_SYNC_LEVEL
- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

AP_PROG_ERROR_NO_TRUNC
Primary return code; the partner TP has issued SEND_ERROR while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued SEND_ERROR. Data sent but not yet received is purged.

AP_PROG_ERROR_TRUNC
Primary return code; the partner TP has issued SEND_ERROR while the conversation was in SEND state. Data was truncated.

AP_SVC_ERROR_NO_TRUNC
Primary return code; the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP was not truncated.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

AP_SVC_ERROR_TRUNC
Primary return code; the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been truncated.

Remarks

While a **POST_ON_RECEIPT** verb is outstanding, the following verbs can be issued on the same conversation:

GET_ATTRIBUTES

GET_TYPE

DEALLOCATE

RECEIVE_AND_WAIT

RECEIVE_IMMEDIATE

REQUEST_TO_SEND

SEND_ERROR

TEST_RTS

TP_ENDED

Issuing any of the following verbs prior to completion of the asynchronous **POST_ON_RECEIPT** verb causes the **POST_ON_RECEIPT** verb to be canceled (the Win32 event is signaled and the primary return code in the verb control block is set to AP_CANCELLED).

DEALLOCATE

RECEIVE_AND_WAIT

RECEIVE_IMMEDIATE

SEND_ERROR

TP_ENDED

PREPARE_TO_RECEIVE

The **PREPARE_TO_RECEIVE** verb changes the state of the conversation for the local TP from SEND to RECEIVE.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **PREPARE_TO_RECEIVE** verb.

```
struct prepare_to_receive {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    primary_rc;
    unsigned short   reserv2;
    unsigned long    secondary_rc;
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    ptr_type;
    unsigned char    locks;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_PREPARE_TO_RECEIVE.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

ptr_type
Supplied parameter. Specifies how to perform the state change.

Use AP_FLUSH to send the contents of the local LU's send buffer to the partner LU (and TP) before changing the conversation's state to RECEIVE.

The AP_SYNC_LEVEL value uses the conversation's synchronization level (established by **ALLOCATE**) to determine how to perform the state change.

If the synchronization level of the conversation is AP_NONE, APPC sends the contents of the local LU's send buffer to the partner TP before changing the conversation's state to RECEIVE. If the synchronization level is AP_CONFIRM_SYNC_LEVEL, APPC sends the contents of the local LU's send buffer and a confirmation request to the partner TP. Upon receiving confirmation from the partner TP, APPC changes the conversation's state to RECEIVE. If, however, the partner TP reports an error, the state changes to RECEIVE or RESET. See the Remarks in this topic.

locks
Supplied parameter. Specifies when APPC should return control to the local TP.

Use this parameter only if **ptr_type** is set to AP_SYNC_LEVEL and the synchronization level of the conversation, established by ALLOCATE, is AP_CONFIRM_SYNC_LEVEL. (Otherwise, the parameter is ignored.)

Use AP_LONG to indicate that APPC returns control to the local TP when the confirmation and subsequent data from the partner TP arrive at the local LU. (This method results in more efficient use of the network but requires a longer time to return control to the local TP.)

Use AP_SHORT to indicate that APPC returns control to the local TP when the confirmation from the partner TP arrives at the local LU.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_P_TO_R_INVALID_TYPE

Secondary return code; the **ptr_type** parameter was not set to a valid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_P_TO_R_NOT_SEND_STATE

Secondary return code; the conversation was not in SEND state.

AP_P_TO_R_NOT_LL_BDY

Secondary return code; the local TP did not finish sending a logical record.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued SEND_ERROR with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY

Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued a SEND_ERROR verb with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

Remarks

Before changing the conversation state, this verb performs the equivalent of one of the following:

- FLUSH, by sending the contents of the local LU's send buffer to the partner LU (and TP).
- CONFIRM, by sending the contents of the local LU's send buffer and a confirmation request to the partner TP.

After this verb has successfully executed, the local TP can receive data.

The conversation must be in SEND state when the TP issues this verb.

State changes, summarized in the following table, are based on the value of **primary_rc**.

primary_rc	**New state**
AP_OK	RECEIVE
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

The conversation does not change to SEND state for the partner TP until the partner TP receives one of the following values through the **what_rcvd** parameter of a subsequent receive verb:

- AP_SEND
- AP_CONFIRM_SEND and replies with CONFIRMED
- AP_DATA_COMPLETE_CONFIRM_SEND and replies with **CONFIRMED**
- AP_DATA_CONFIRM_SEND and replies with **CONFIRMED**

The receive verbs are RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2), RECEIVE_IMMEDIATE, and RECEIVE_AND_WAIT.

RECEIVE_ALLOCATE

The **RECEIVE_ALLOCATE** verb is issued by the invoked TP to confirm that the invoked TP is ready to begin a conversation with the invoking TP that issued ALLOCATE or MC_ALLOCATE.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **RECEIVE_ALLOCATE** verb.

```
struct receive_allocate {
    unsigned short   opcode;
    unsigned char    opext;
    unsigned char    reserv2;
    unsigned short   primary_rc;
    unsigned long    secondary_rc;
    unsigned char    tp_name[64];
    unsigned char    tp_id[8];
    unsigned long    conv_id;
    unsigned char    sync_level;
    unsigned char    conv_type;
    unsigned char    user_id[10];
    unsigned char    lu_alias[8];
    unsigned char    plu_alias[8];
    unsigned char    mode_name[8];
    unsigned char    reserv3[2];
    unsigned long    conv_group_id;
    unsigned char    fqplu_name[17];
    unsigned char    pip_incoming;
    unsigned char    syncpoint_rqd;
    unsigned char    reserv4[3];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_RECEIVE_ALLOCATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_name
Supplied parameter. Provides the name of the local TP. The value of **tp_name** must match the TP name configured through registry or environment variables. APPC matches the **RECEIVE_ALLOCATE** verb's **tp_name** parameter with the TP name specified by the incoming allocate, which is generated by MC_ALLOCATE or ALLOCATE in the invoking TP.

This parameter is a 64-byte EBCDIC character string and is case-sensitive. The **tp_name** parameter can consist of characters from the type AE EBCDIC character set:

- Uppercase and lowercase letters
- Numerals 0 through 9
- Special characters $, #, and period (.)

If **tp_name** is fewer than 64 bytes, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention is that a service TP name can have up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The other characters are from the type AE EBCDIC character set.

tp_id

Returned parameter. Identifies the local TP.

conv_id

Returned parameter. Provides the conversation identifier. It identifies the conversation APPC has established between the two partner TPs.

sync_level

Returned parameter. Specifies the synchronization level of the conversation. It determines whether the TPs can request confirmation of receipt of data and confirm receipt of data.

- AP_NONE specifies that confirmation processing will not be used in this conversation.
- AP_CONFIRM_SYNC_LEVEL specifies that the TPs can use confirmation processing in this conversation.
- AP_SYNCPT specifies that TPs can use Sync Point Level 2 confirmation processing in this conversation.

conv_type

Returned parameter. Specifies the type of conversation chosen by the partner TP, using MC_ALLOCATE or ALLOCATE. The following are possible values:

AP_BASIC_CONVERSATION

AP_MAPPED_CONVERSATION

user_id

Returned parameter. Provides the user identifier specified by the partner TP, using **MC_ALLOCATE** or **ALLOCATE** (if the partner TP set the **MC_ALLOCATE** or **ALLOCATE** verb's security parameter to AP_PGM or AP_SAME). It is a type AE EBCDIC character string.

lu_alias

Returned parameter. Provides the alias by which the local LU is known to the local TP. It is an ASCII character string.

plu_alias

Returned parameter. Provides the alias by which the partner LU (which initiated the incoming allocate) is known to the local TP. It is an ASCII character string.

mode_name

Returned parameter. Provides the mode name specified by **MC_ALLOCATE** or **ALLOCATE** in the partner TP. It is the name of a set of networking characteristics defined during configuration. The **mode_name** is a type A EBCDIC character string.

reserv3

A reserved field.

conv_group_id

Conversation group identifier.

fqplu_name

This returned parameter provides the fully qualified LU name.

pip_incoming

This optional supplied and returned parameter is applicable only if Sync Point services are required.

For the supplied parameter:

AP_YES if TP does accept PIP data.

AP_NO if TP does not accept PIP data.

For the returned parameter:

AP_YES if PIP data is available.

AP_NO if PIP data is not available.

syncpoint_rqd

This parameter indicates if Sync Point services are required.

AP_YES if Sync Point is required.

AP_NO if Sync Point is not required.

reserv4

A reserved field.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_UNDEFINED_TP_NAME

Secondary return code; the TP name was not configured correctly.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_ALLOCATE_NOT_PENDING

Secondary return code; APPC did not find an incoming allocate (from the invoking TP) to match the value of **tp_name**, supplied by **RECEIVE_ALLOCATE**. **RECEIVE_ALLOCATE** waited for the incoming allocate and eventually timed out.

AP_INVALID_PROCESS

Secondary return code; the process issuing **RECEIVE_ALLOCATE** was different from the one started by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

This must be the first APPC verb issued by the invoked TP. The initial state is RESET. If the verb executes successfully (**primary_rc** is AP_OK), the state changes to RECEIVE.

In response to this verb, APPC establishes a conversation between the two TPs and generates a TP identifier for the invoked TP and a conversation identifier. These identifiers are required parameters for subsequent APPC verbs.

If the invoked TP issues **RECEIVE_ALLOCATE** and a corresponding incoming allocate (resulting from MC_ALLOCATE or ALLOCATE issued by the invoking TP) is not present, the invoked TP waits until the incoming allocate arrives or the verb times out. The time-out value is set by the system administrator.

RECEIVE_AND_POST

The **RECEIVE_AND_POST** verb receives application data and status information asynchronously. This allows the local TP to proceed with processing while data is still arriving at the local LU.

RECEIVE_AND_POST is only supported under the Microsoft® Windows NT®, Microsoft® Windows® 95, and OS/2 operating systems. For similar functionality under the Windows version 3.*x* graphical environment, use RECEIVE_AND_WAIT in conjunction with WinAsyncAPPC. Specifically, while an asynchronous **RECEIVE_AND_POST** is outstanding, the following verbs can be issued on the same conversation:

DEALLOCATE (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)

GET_ATTRIBUTES

GET_TYPE

REQUEST_TO_SEND

SEND_ERROR

TEST_RTS

TP_ENDED

This allows an application to use an asynchronous **RECEIVE_AND_POST** to receive data. While the **RECEIVE_AND_POST** is outstanding, it can still use **SEND_ERROR** and **REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support. For information on how a TP receives data and how to use this verb, see Remarks in this topic.

The following structure describes the verb control block used by the **RECEIVE_AND_POST** verb.

```
struct receive_and_post {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       fill;
    unsigned char       rts_rcvd;
    unsigned char       reserv4;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char FAR * sema;
    unsigned char       reserv5;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_RECEIVE_AND_POST.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd
Returned parameter. Indicates whether data or conversation status was received. Possible values are listed following the **Members** section.

rtn_status
Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

- AP_NO specifies that indicators should be returned individually on separate invocations of the verb.
- AP_YES specifies that indicators should be returned together, provided both are available. Both can be returned when:

  The receive buffer is large enough to hold all of the data that precedes the status indicator.

  The **fill** parameter specifies BUFFER or LL, and the data is the last logical record before the status indicator.

fill
Supplied parameter. Specifies how the local TP receives data.

Use AP_BUFFER to indicate that the local TP receives data until the number of bytes specified by **max_len** is reached or until end of data. Data is received without regard for the logical-record format.

Use AP_LL to indicate that data is received in logical-record format. The data received can be:

- A complete logical record.
- A **max_len** byte portion of a logical record.
- The end of a logical record**.**

rts_rcvd
Returned parameter. Indicates whether the partner TP issued REQUEST_TO_SEND. Possible values are:

- AP_YES indicates that the partner TP issued **REQUEST_TO_SEND**, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **REQUEST_TO_SEND**.

max_len
Supplied parameter. Specifies the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

The value must not exceed the length of the buffer to contain the received data. The offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

dlen
Returned parameter. Specifies the number of bytes of data received. Data is stored in the buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr
Supplied parameter. Provides the address of the buffer to contain the data received by the local LU.

For the Windows NT and Windows 95 operating systems, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the **DosAllocSeg** function with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

sema
Supplied parameter. Provides the address of the semaphore that APPC is to clear when the asynchronous receiving operation is finished. On OS/2, the **sema** parameter is either a RAM or system semaphore. On Windows NT and Windows 95, the **sema** parameter is an event handle obtained by calling either the **CreateEvent** or **OpenEvent** Win32® function.

Values returned by the what_rcvd parameter

- AP_CONFIRM_DEALLOCATE indicates that the partner TP issued DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL. The conversation's synchronization level, established by ALLOCATE, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP issued PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL. The conversation's synchronization level, established by **ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value,

the local TP normally issues **CONFIRMED**, and begins to send data.

- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP issued CONFIRM. Upon receiving this value, the local TP normally issues **CONFIRMED**.
- AP_DATA indicates that this value can be returned by **RECEIVE_AND_POST** if **fill** is set to AP_BUFFER. The local TP received data until **max_len** or the end of the data was reached. For more information, see Remarks in this topic.
- AP_DATA_COMPLETE indicates, for **RECEIVE_AND_POST**, that the local TP has received a complete data record or the last part of a data record.

  For **RECEIVE_AND_POST** with **fill** set to AP_LL, this value indicates that the local TP has received a complete logical record or the end of a logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_AND_POST** or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data.

  Otherwise, the local TP examines status information.

  If **primary_rc** contains AP_OK and **what_rcvd** contains AP_SEND, AP_CONFIRM_SEND, AP_CONFIRM_DEALLOCATE, or AP_CONFIRM_WHAT_RECEIVED, see the description of the value (in this section) for the next action the local TP normally takes.

  If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to the DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates, for **RECEIVE_AND_POST**, that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record).

  For **RECEIVE_AND_POST** with **fill** set to AP_LL, this value indicates that the local TP has received an incomplete logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_AND_POST** (or issues another receive verb) to receive the next part of the record.

- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, that the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses SEND_DATA to begin sending data.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.
AP_DATA_COMPLETE_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.
AP_DATA_COMPLETE_CONFIRM
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.
AP_DATA_COMPLETE_CONFIRM_DEALL
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.
AP_DATA_SEND
Primary return code; this is a combination of AP_DATA and AP_SEND.
AP_DATA_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_SEND.
AP_DATA_CONFIRM
Primary return code; this is a combination of AP_DATA and AP_CONFIRM.
AP_DATA_CONFIRM_DEALLOCATE
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_DEALLOCATE.
AP_DEALLOC_NORMAL
Primary return code; the partner TP issued DEALLOCATE with **dealloc_type** set to AP_FLUSH or AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE.

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_INVALID_SEMAPHORE_HANDLE

Secondary return code; the address of the RAM semaphore or system semaphore handle was invalid.

> APPC cannot trap all invalid semaphore handles. If the TP passes a bad RAM semaphore handle, a protection violation results.

AP_RCV_AND_POST_BAD_FILL

Secondary return code; the **fill** parameter was set to an invalid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_AND_POST_BAD_STATE

Secondary return code; the conversation was not in RECEIVE or SEND state when the TP issued this verb.

AP_RCV_AND_POST_NOT_LL_BDY

Secondary return code; the conversation was in SEND state; the TP began but did not finish sending a logical record.

AP_CANCELED
Primary return code; the local TP issued one of the following verbs, which canceled **RECEIVE_AND_POST**:

DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER

SEND_ERROR

TP_ENDED

Issuing one of these verbs causes the semaphore to be cleared.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the

partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY

Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY

Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC
Primary return code; the partner TP issued SEND_ERROR with **err_type** set to AP_PROG while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_PROG_ERROR_TRUNC
Primary return code; in SEND state, after sending an incomplete logical record, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. The local TP may have received the first part of the logical record through a receive verb.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_NO_TRUNC
Primary return code; while in SEND state, the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC. Data was not truncated.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

AP_SVC_ERROR_TRUNC
Primary return code; in SEND state, after sending an incomplete logical record, the partner TP (or partner LU) issued **SEND_ERROR**. The local TP may have received the first part of the logical record.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received can be:

   - One logical record.
   - A buffer of data received independent of its logical-record format.

   The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it. The receive verbs are **RECEIVE_AND_POST** (Windows NT, Windows 95, and OS/2), RECEIVE_AND_WAIT, and RECEIVE_IMMEDIATE.

2. The local TP issues the receive verb again. This has one of the following effects:

- If the partner TP has sent more data, the local TP begins to receive a new unit of data.
- If the partner TP has finished sending data or is waiting for confirmation, status information (available through **what_rcvd**) indicates the next action the local TP normally takes.

The following procedure shows tasks performed by the local TP in using **RECEIVE_AND_POST**.

To use RECEIVE_AND_POST

1. For the Windows NT and Windows 95 operating systems, the TP retrieves the WinAsyncAPPC message number by calling the **RegisterWindowMessage** API or allocating a semaphore. The **sema** field should be set to NULL if the application expects to be notified through the Windows message mechanism.

   APPC sends the Windows message or clears the semaphore when the local TP finishes receiving data.

   For the OS/2 operating system, the TP uses the **DosSemSet** function to set the semaphore pointed to by **sema**.

   The semaphore will remain set while the local TP receives data asynchronously. APPC will clear the semaphore when the local TP finishes receiving data.

2. The TP issues RECEIVE_AND_POST.
3. The TP checks the value of primary_rc.

   If **primary_rc** is AP_OK, the receive buffer (pointed to by **dptr**) is asynchronously receiving data from the partner TP. While receiving data asynchronously, the local TP can:

   - Perform tasks not related to this conversation.
   - Issue REQUEST_TO_SEND.
   - Gather information about this conversation by issuing GET_TYPE, GET_ATTRIBUTES, or TEST_RTS.
   - Prematurely cancel **RECEIVE_AND_POST** by issuing DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER; SEND_ERROR; or TP_ENDED.

   If, however, **primary_rc** is not AP_OK, **RECEIVE_AND_POST** has failed. In this case, the local TP does not perform the next two tasks.

4. For the Windows NT and Windows 95 operating systems, when the TP finishes receiving data asynchronously, APPC issues the WinAsyncAPPC Windows message or clears the semaphore.

   For the OS/2 operating system, the TP uses the **DosSemWait** function to wait for APPC to clear the semaphore pointed to by **sema**. When the TP finishes receiving data asynchronously, APPC clears the semaphore. To prevent the local TP from waiting, have it test the semaphore (invoking **DosSemWait** with **Timeout** set to zero) until APPC clears the semaphore.

5. The TP checks the new value of **primary_rc**.

   If **primary_rc** is AP_OK, the local TP can examine the other returned parameters and manipulate the asynchronously received data.

   If **primary_rc** is not AP_OK, only **secondary_rc** and **rts_rcvd** (request-to-send received) are meaningful.

Conversation State Effects

The conversation must be in RECEIVE or SEND state when the TP issues this verb.

Issuing **RECEIVE_AND_POST** while the conversation is in SEND state has the following effects:

- The local LU sends the information in its send buffer and a SEND indicator to the partner TP.
- The conversation changes to PENDING_POST state; the local TP is ready to receive information from the partner TP asynchronously.

The conversation changes states twice:

- Upon initial return of the verb, if **primary_rc** contains AP_OK, the conversation changes to PENDING_POST state.
- After completion of the verb, the state changes depending on the value of the following:

  The **primary_rc** parameter

  The **what_rcvd** parameter if **primary_rc** is AP_OK

The following table shows the new state associated with each value of **what_rcvd** when **primary_rc** is AP_OK.

what_rcvd	New state
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	RECEIVE
AP_DATA_COMPLETE	RECEIVE
AP_DATA_INCOMPLETE	RECEIVE
AP_SEND	SEND
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table shows the new state associated with each value of **primary_rc** other than AP_OK.

primary_rc	New state
AP_CANCELED	No change
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_PROG_ERROR_NO_TRUNC	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_NO_TRUNC	RECEIVE
AP_PROG_ERROR_TRUNC	RECEIVE
AP_SVC_ERROR_TRUNC	RECEIVE

End of Data for a Basic Conversation

If the local TP issues **RECEIVE_AND_POST** and sets **fill** to AP_BUFFER, the receipt of data ends when **max_len** or the end of the data is reached. The end of the data is indicated by either **primary_rc** with a value other than AP_OK (for example, AP_DEALLOC_NORMAL), or by **what_rcvd** with one of the following values:

AP_SEND

AP_CONFIRM_SEND

AP_CONFIRM_DEALLOCATE

AP_CONFIRM_WHAT_RECEIVED

AP_DATA_CONFIRM_SEND

AP_DATA_CONFIRM_DEALLOCATE

AP_DATA_CONFIRM

To determine if the end of the data has been reached, the local TP reissues **RECEIVE_AND_POST**. If the new **primary_rc** contains AP_OK and **what_rcvd** contains AP_DATA, the end of the data has not been reached. If, however, the end of the data has been reached, **primary_rc** or **what_rcvd** will indicate the cause of the end of the data.

Troubleshooting

The local TP can wait indefinitely if one of the following situations occurs:

- For the Windows NT and Windows 95 operating systems, the local TP issues a **RECEIVE_AND_POST** request, but either the partner TP has not sent data or the initial **primary_rc** is not AP_OK.

- For the OS/2 operating system, the local TP issues a **DosSemWait** function call, but either the partner TP has not sent data or the initial **primary_rc** is not AP_OK.

This is because APPC will not issue the Windows message or clear the semaphore.

When a condition resulting in one of the following **primary_rc** parameters occurs, APPC does not clear the semaphore:

AP_INVALID_SEMAPHORE_HANDLE

AP_INVALID_VERB_SEGMENT

AP_STACK_TOO_SMALL

To test **what_rcvd**, issue **RECEIVE_AND_POST** with **max_len** set to zero, so that the local TP can determine whether the partner TP has data to send, seeks confirmation, or has changed the conversation state.

RECEIVE_AND_WAIT

The **RECEIVE_AND_WAIT** verb receives any data that is currently available from the partner TP. If no data is currently available, the local TP waits for data to arrive.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call. To allow full use to be made of the asynchronous support, asynchronously issued **RECEIVE_AND_WAIT** verbs have been altered to act like RECEIVE_AND_POST verbs. Specifically, while an asynchronous **RECEIVE_AND_WAIT** is outstanding, the following verbs can be issued on the same conversation:

DEALLOCATE (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)

GET_ATTRIBUTES

GET_TYPE

REQUEST_TO_SEND

SEND_ERROR

TEST_RTS

TP_ENDED

This allows an application, and in particular, a 5250 emulator, to use an asynchronous **RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_WAIT** is outstanding, it can still use **SEND_ERROR** and **REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support.

The following structure describes the verb control block used by the **RECEIVE_AND_WAIT** verb.

```
struct receive_and_wait {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       fill;
    unsigned char       rts_rcvd;
    unsigned char       reserv4;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char       reserv5[5];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_RECEIVE_AND_WAIT.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Specifies the conversation identifier.

The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd
Returned parameter. Indicates whether data or conversation status was received. Possible values are listed in the table following the Members section.

rtn_status
Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

- AP_NO specifies that indicators should be returned individually on separate invocations of the verb.
- AP_YES specifies that indicators should be returned together, provided both are available. Both can be returned when:

  The receive buffer is large enough to hold all of the data that precedes the status indicator.

  The **fill** parameter specifies BUFFER or LL, and the data is the last logical record before the status indicator.

fill
Supplied parameter. Used in a basic conversation to specify how the local TP receives data. The following are allowed values:

- AP_BUFFER specifies that the local TP receives data until the number of bytes specified by **max_len** is reached or until the end of the data. Data is received without regard for the logical-record format.
- AP_LL specifies that data is received in logical-record format. The data received can be a complete logical record, a **max_len** byte portion of a logical record, or the end of a logical record.

rts_rcvd
Returned parameter. Contains the request-to-send indicator. Possible values are:

- AP_YES indicates that the partner TP has issued REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **REQUEST_TO_SEND**.

max_len
Supplied parameter. Indicates the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

For the Microsoft® Windows NT® and Windows 95 operating systems and the Windows graphical environment, this value must not exceed the length of the buffer to contain the received data.

For the OS/2 operating system, the offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

By issuing **RECEIVE_AND_WAIT** with **max_len** set to zero, the local TP can determine whether the partner TP has data to send, seeks confirmation, or has changed the conversation state.

dlen
Returned parameter. Indicates the number of bytes of data received. Data is stored in the buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr
Supplied parameter. Provides the address of the buffer to contain the data received by the local TP.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the **DosAllocSeg** function with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

For the Windows environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

Values returned by the what_rcvd member

- AP_CONFIRM_DEALLOCATE indicates that the partner TP has issued DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP has issued PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this

value, the local TP normally issues **CONFIRMED** and begins to send data.

- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP has issued CONFIRM. Upon receiving this value, the local TP normally issues **CONFIRMED**.
- AP_DATA can be returned in a basic conversation by **RECEIVE_AND_WAIT** if **fill** is set to AP_BUFFER. The local TP received data until **max_len** or end of data was reached. For more information, see "**RECEIVE_AND_WAIT** End of Data" at the end of this topic.
- AP_DATA_COMPLETE indicates, for **RECEIVE_AND_WAIT**, that the local TP has received a complete data record or the last part of a data record.

  For **RECEIVE_AND_WAIT** with **fill** set to AP_LL, this value indicates that the local TP has received a complete logical record or the end of a logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_AND_WAIT** or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data.

  Otherwise, the local TP examines status information, if **primary_rc** contains AP_OK and **what_rcvd** contains AP_SEND, AP_CONFIRM_SEND, AP_CONFIRM_DEALLOCATE, or AP_CONFIRM_WHAT_RECEIVED.

  See Return Codes in this topic for the next action the local TP normally takes.

  If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates, for **RECEIVE_AND_WAIT**, that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record).

  For **RECEIVE_AND_WAIT** with **fill** set to AP_LL, this value indicates that the local TP has received an incomplete logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_AND_WAIT** (or issues another receive verb) to receive the next part of the record.

- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, that the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses SEND_DATA to begin sending data.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.
AP_DATA_COMPLETE_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.
AP_DATA_COMPLETE_CONFIRM
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.
AP_DATA_COMPLETE_CONFIRM_DEALL
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.
AP_DATA_SEND
Primary return code; this is a combination of AP_DATA and AP_SEND.
AP_DATA_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_SEND.
AP_DATA_CONFIRM
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_WHAT_RECEIVED.
AP_DATA_CONFIRM_DEALLOCATE
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_DEALLOCATE.
AP_DEALLOC_NORMAL
Primary return code; the partner TP has deallocated the conversation without requesting confirmation and issued DEALLOCATE with **dealloc_type** set to one of the following:

AP_CONFIRM_SYNC_LEVEL

AP_FLUSH

AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_RCV_AND_WAIT_BAD_FILL

Secondary return code; for basic conversations, **fill** was set to an invalid value.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_AND_WAIT_BAD_STATE

Secondary return code; the conversation was not in RECEIVE or SEND state when the TP issued this verb.

AP_RCV_AND_WAIT_NOT_LL_BDY

Secondary return code; for basic conversations, the conversation was in SEND state; the TP began but did not finish sending a logical record.

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY

Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY

Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC

Primary return code; the partner TP has issued SEND_ERROR with **err_type** set to AP_PROG while the conversation was in

SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_PROG_ERROR_TRUNC
Primary return code; in SEND state, after sending an incomplete logical record, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. The local TP may have received the first part of the logical record through a receive verb.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP has issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_NO_TRUNC
Primary return code; while in SEND state, the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC. Data was not truncated.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

AP_SVC_ERROR_NO_TRUNC
Primary return code; while in SEND state, after sending an incomplete logical record, the partner TP (or partner LU) issued **SEND_ERROR**. The local TP may have received the first part of the logical record.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received can be:

   - One logical record.
   - A buffer of data received independent of its logical-record format.

   The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it.

   The receive verbs are RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems), **RECEIVE_AND_WAIT**, and RECEIVE_IMMEDIATE.

2. The local TP issues the receive verb again. This has one of the following effects:

   - If the partner TP has sent more data, the local TP begins to receive a new unit of data.
   - If the partner TP has finished sending data or is waiting for confirmation, status information (available through the **what_rcvd** parameter) indicates the next action the local TP normally takes.

The conversation must be in RECEIVE or SEND state when the TP issues this verb.

Issuing the Verb in SEND State

Issuing **RECEIVE_AND_WAIT** while the conversation is in SEND state has the following effects:

- The local LU sends the information in its send buffer and a SEND indicator to the partner TP.
- The conversation changes to RECEIVE state; the local TP waits for the partner TP to send data.

State Change

The new conversation state is determined by the following factors:

- The state the conversation is in when the TP issues the verb.
- The **primary_rc** parameter.
- The **what_rcvd** parameter if **primary_rc** contains AP_OK.

Verb Issued in SEND State

The following table details the state changes when **RECEIVE_AND_WAIT** is issued in SEND state and **primary_rc** is AP_OK.

what_rcvd	New state
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	RECEIVE
AP_DATA_COMPLETE	RECEIVE
AP_DATA_INCOMPLETE	RECEIVE
AP_SEND	No change
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when **RECEIVE_AND_WAIT** is issued in SEND state and **primary_rc** is not AP_OK.

primary_rc	New state
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_PROG_ERROR_NO_TRUNC	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_NO_TRUNC	RECEIVE

Verb Issued in RECEIVE State

The following table details the state changes when **RECEIVE_AND_WAIT** is issued in RECEIVE state and **primary_rc** is AP_OK.

what_rcvd	New state
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND

AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	No change
AP_DATA_COMPLETE	No change
AP_DATA_INCOMPLETE	No change
AP_SEND	SEND
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when **RECEIVE_AND_WAIT** is issued in RECEIVE state and **primary_rc** is not AP_OK.

primary_rc	**New state**
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	No change
AP_PROG_ERROR_NO_TRUNC	No change
AP_SVC_ERROR_PURGING	No change
AP_SVC_ERROR_NO_TRUNC	No change
AP_PROG_ERROR_TRUNC	No change
AP_SVC_ERROR_TRUNC	No change

RECEIVE_AND_WAIT End of Data

In basic conversations, if the local TP issues **RECEIVE_AND_WAIT** and sets **fill** to AP_BUFFER, the receipt of the data ends when **max_len** or the end of the data is reached. The end of the data is indicated by either:

- A **primary_rc** parameter with a value other than AP_OK (for example, AP_DEALLOC_NORMAL).
- A **what_rcvd** parameter with one of the following values:

  AP_SEND

  AP_CONFIRM_SEND

  AP_CONFIRM_DEALLOCATE

  AP_CONFIRM_WHAT_RECEIVED

  AP_DATA_CONFIRM_SEND

  AP_DATA_CONFIRM_DEALLOCATE

  AP_DATA_CONFIRM

To determine if the end of the data has been reached, the local TP reissues **RECEIVE_AND_WAIT**. If the new **primary_rc** contains AP_OK and **what_rcvd** contains AP_DATA, the end of the data has not been reached. If, however, the end of the data has been reached, **primary_rc** or **what_rcvd** will indicate the cause of the end of the data.

RECEIVE_AND_WAIT waits for data or an indicator to be sent by the partner TP. If you need the local TP to operate continuously, use RECEIVE_IMMEDIATE instead.

RECEIVE_IMMEDIATE

The **RECEIVE_IMMEDIATE** verb receives any data currently available from the partner TP. If no data is available, the local TP does not wait. To avoid blocking the conversation, the Microsoft® Windows NT®, Microsoft® Windows® 95, and Windows version 3.*x* systems can issue RECEIVE_AND_WAIT in conjunction with WinAsyncAPPC. For OS/2 systems, use RECEIVE_AND_POST.

The following structure describes the verb control block used by the **RECEIVE_IMMEDIATE** verb.

```
struct receive_immediate {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      what_rcvd;
    unsigned char       rtn_status;
    unsigned char       fill;
    unsigned char       rts_rcvd;
    unsigned char       reserv4;
    unsigned short      max_len;
    unsigned short      dlen;
    unsigned char FAR * dptr;
    unsigned char       reserv5[5];
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_RECEIVE_IMMEDIATE.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

what_rcvd
Returned parameter. Contains information received with the incoming data. Possible values are listed in the table following the Members section.

rtn_status
Supplied parameter. Indicates whether both data and conversation status indicators should be returned within one API call.

Use AP_NO to specify that indicators should be returned individually on separate invocations of the verb.

Use AP_YES to specify that indicators should be returned together, provided both are available. Both can be returned when:

- The receive buffer is large enough to hold all of the data that precedes the status indicator.
- The **fill** parameter specifies either BUFFER or LL, and the data is the last logical record before the status indicator.

fill
Supplied parameter. Specifies the manner in which the local TP receives data. It is used only for **RECEIVE_IMMEDIATE**.

Use AP_BUFFER to indicate that the local TP receives data until the number of bytes specified by **max_len** is reached or until

end of data. Data is received without regard for the logical-record format.

Use AP_LL to indicate that data is received in logical-record format. The data received can be a complete logical record, a **max_len** byte portion of a logical record, or the end of a logical record.

rts_rcvd
Returned parameter. Contains the request-to-send indicator. Possible values are:

- AP_YES indicates that the partner TP has issued REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state.
- AP_NO indicates that the partner TP has not issued **REQUEST_TO_SEND**.

max_len
Supplied parameter. Indicates the maximum number of bytes of data the local TP can receive. The range is from 0 through 65535.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, this value must not exceed the length of the buffer to contain the received data.

For the OS/2 operating system, the offset of **dptr** plus the value of **max_len** must not exceed the size of the data segment.

By issuing **RECEIVE_IMMEDIATE** with **max_len** set to zero, the local TP can determine whether the partner TP has data to send, seeks confirmation, or has changed the conversation state.

dlen
Returned parameter. Provides the number of bytes of data received. Data is stored in a buffer specified by **dptr**. A length of zero indicates that no data was received.

dptr
Supplied parameter. Address of the buffer to contain the data received by the local TP.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

Values returned by the what_rcvd member

- AP_CONFIRM_DEALLOCATE indicates that the partner TP has issued DEALLOCATE with **dealloc_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues CONFIRMED.
- AP_CONFIRM_SEND indicates that the partner TP has issued PREPARE_TO_RECEIVE with **ptr_type** set to AP_SYNC_LEVEL, and the conversation's synchronization level, established by **ALLOCATE**, is AP_CONFIRM_SYNC_LEVEL. Upon receiving this value, the local TP normally issues **CONFIRMED**, and begins to send data.
- AP_CONFIRM_WHAT_RECEIVED indicates that the partner TP has issued CONFIRM. Upon receiving this value, the local TP normally issues **CONFIRMED**.
- AP_DATA is returned for basic conversations by **RECEIVE_IMMEDIATE** if **fill** is set to AP_BUFFER. The local TP received data until **max_len** or end of data was reached. For more information, see "**RECEIVE_IMMEDIATE** End of Data" at the end of this topic.
- AP_DATA_COMPLETE indicates, for **RECEIVE_IMMEDIATE** with **fill** set to AP_LL in basic conversations, that the local TP has received a complete logical record or the end of a logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_IMMEDIATE** or issues another receive verb. If the partner TP has sent more data, the local TP begins to receive a new unit of data.

  Otherwise, the local TP examines status information if **primary_rc** contains AP_OK and **what_rcvd** contains any of these values:

  AP_SEND

  AP_CONFIRM_SEND

  AP_CONFIRM_DEALLOCATE

  AP_CONFIRM_WHAT_RECEIVED

  See the description of the value in Return Codes in this topic for the next action the local TP normally takes.

If **primary_rc** contains AP_DEALLOC_NORMAL, the conversation has been deallocated in response to DEALLOCATE issued by the partner TP.

- AP_DATA_INCOMPLETE indicates for **RECEIVE_IMMEDIATE** in mapped conversations that the local TP has received an incomplete data record. The **max_len** parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first receive verb to read the record).

  For **RECEIVE_IMMEDIATE** with **fill** set to AP_LL in basic conversations, this value indicates that the local TP has received an incomplete logical record.

  Upon receiving this value, the local TP normally reissues **RECEIVE_IMMEDIATE** (or issues another receive verb) to receive the next part of the record.

- AP_NONE indicates that the TP did not receive data or conversation status indicators.
- AP_SEND indicates, for the partner TP, the conversation has entered RECEIVE state. For the local TP, the conversation is now in SEND state. Upon receiving this value, the local TP normally uses SEND_DATA to begin sending data.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

When **rtn_status** is AP_YES, the preceding return code or one of the following return codes can be returned.

AP_DATA_COMPLETE_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_SEND.
AP_DATA_COMPLETE_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_SEND.
AP_DATA_COMPLETE_CONFIRM
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_WHAT_RECEIVED.
AP_DATA_COMPLETE_CONFIRM_DEALL
Primary return code; this is a combination of AP_DATA_COMPLETE and AP_CONFIRM_DEALLOCATE.
AP_DATA_SEND
Primary return code; this is a combination of AP_DATA and AP_SEND.
AP_DATA_CONFIRM_SEND
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_SEND.
AP_DATA_CONFIRM
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_WHAT_RECEIVED.
AP_DATA_CONFIRM_DEALLOCATE
Primary return code; this is a combination of AP_DATA and AP_CONFIRM_DEALLOCATE.
AP_UNSUCCESSFUL
Primary return code; no data is immediately available from the partner TP.
AP_DEALLOC_NORMAL
Primary return code; the partner TP has deallocated the conversation without requesting confirmation. The partner TP issued DEALLOCATE with **dealloc_type** set to one of the following:

- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

If **rtn_status** is AP_YES, examine **what_rcvd** also.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_RETURN_STATUS_WITH_DATA

Secondary return code; the specified **rtn_status** value was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_RCV_IMMD_BAD_FILL

Secondary return code for a basic conversation; the **fill** parameter was set to an invalid value.

AP_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

AP_RCV_IMMD_BAD_STATE

Secondary return code; the conversation was not in RECEIVE state.

AP_ALLOCATION_ERROR

Secondary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it can indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY

Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY

Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_NO_TRUNC

Primary return code; the partner TP has issued SEND_ERROR with **err_type** set to AP_PROG while the conversation was in SEND state. Data was not truncated.

AP_PROG_ERROR_PURGING

Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_PROG_ERROR_TRUNC

Primary return code; in SEND state, after sending an incomplete logical record, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. The local TP may have received the first part of the logical record through a receive verb.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY

Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG

Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP has issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC

Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_NO_TRUNC
Primary return code; while in SEND state, the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC. Data was not truncated.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

AP_SVC_ERROR_TRUNC
Primary return code; in SEND state, after sending an incomplete logical record, the partner TP (or partner LU) issued **SEND_ERROR**. The local TP may have received the first part of the logical record.

Remarks

The local TP receives data through the following process:

1. The local TP issues a receive verb until it finishes receiving a complete unit of data. The data received can be:

   - One logical record.
   - A buffer of data received independent of its logical-record format.

   The local TP may need to issue the receive verb several times in order to receive a complete unit of data. After a complete unit of data has been received, the local TP can manipulate it.

   The receive verbs are RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems), RECEIVE_AND_WAIT, and **RECEIVE_IMMEDIATE**.

2. The local TP issues the receive verb again. This has one of the following effects:

   - If the partner TP has sent more data, the local TP begins to receive a new unit of data.
   - If the partner TP has finished sending data or is waiting for confirmation, status information (available through **what_rcvd**) indicates the next action the local TP normally takes.

The conversation must be in RECEIVE state when the TP issues this verb.

The new state is determined by **primary_rc**. If **primary_rc** is AP_OK, the new state is determined by **what_rcvd**.

The following table details the state changes when the **primary_rc** is AP_OK.

what_rcvd	**New state**
AP_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_DATA_COMPLETE_CONFIRM_DEALL	CONFIRM_DEALLOCATE
AP_DATA_CONFIRM_DEALLOCATE	CONFIRM_DEALLOCATE
AP_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_COMPLETE_CONFIRM_SEND	CONFIRM_SEND
AP_DATA_CONFIRM_SEND	CONFIRM_SEND
AP_CONFIRM_WHAT_RECEIVED	CONFIRM
AP_DATA_COMPLETE_CONFIRM	CONFIRM
AP_DATA_CONFIRM	CONFIRM
AP_DATA	No change
AP_DATA_COMPLETE	No change
AP_DATA_INCOMPLETE	No change
AP_SEND	SEND
AP_DATA_COMPLETE_SEND	SEND_PENDING

The following table details the state changes when the **primary_rc** is not AP_OK.

primary_rc	**New state**
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET

AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	No change
AP_PROG_ERROR_NO_TRUNC	No change
AP_SVC_ERROR_PURGING	No change
AP_SVC_ERROR_NO_TRUNC	No change
AP_PROG_ERROR_TRUNC	No change
AP_SVC_ERROR_TRUNC	No change
AP_UNSUCCESSFUL	No change

RECEIVE IMMEDIATE End of Data

In basic conversations, if the local TP issues **RECEIVE_IMMEDIATE** and sets **fill** to AP_BUFFER, the receipt of the data ends when **max_len** or the end of the data is reached. The end of the data is indicated by either:

- A **primary_rc** parameter with a value other than AP_OK (for example, AP_DEALLOC_NORMAL).
- A **what_rcvd** parameter with one of the following values:

  AP_SEND

  AP_CONFIRM_SEND

  AP_CONFIRM_DEALLOCATE

  AP_CONFIRM_WHAT_RECEIVED

  AP_DATA_CONFIRM_SEND

  AP_DATA_CONFIRM_DEALLOCATE

  AP_DATA_CONFIRM

To determine if the end of the data has been reached, the local TP reissues **RECEIVE_IMMEDIATE**. If the new **primary_rc** parameter contains AP_OK and **what_rcvd** contains AP_DATA, the end of the data has not been reached. If, however, the end of the data has been reached, **primary_rc** or **what_rcvd** will indicate the cause of the end of the data.

RECEIVE_LOG_DATA

The **RECEIVE_LOG_DATA** verb allows the user to register to receive the log data associated with an inbound Function Management Header 7 (FMH7) error report. The verb passes a buffer to APPC, and any log data received is placed in that buffer. APPC continues to use this buffer as successive FMH7s arrive until it is provided with another one (that is, until the TP issues another **RECEIVE_LOG_DATA** specifying a different buffer or no buffer at all). This verb is only supported on Microsoft® Windows NT® and Microsoft® Windows® 95 by Microsoft® SNA Server version 3.0 with Service Pack 1 or later and by SNA Server version 4.0.

Note that the TP itself is responsible for allocating and freeing the buffer. After the buffer has been passed to APPC, the TP should either issue another **RECEIVE_LOG_DATA** specifying a new buffer or a zero-length buffer, or wait until the conversation has finished before freeing the original buffer.

When an FMH7 is received, APPC copies any associated error log general data stream (GDS) into the buffer. If there is no associated error log variable, the buffer is zeroed out. It is up to the TP to check the buffer whenever a return code from a receive verb indicates that an error has been received.

The following structure describes the verb control block used by the **RECEIVE_LOG_DATA** verb.

```
struct receive_log_data {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv1;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned short      log_dlen;
    unsigned char FAR * log_dptr;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_RECEIVE_LOG_DATA.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv1
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

log_dlen
Supplied parameter. Specifies the maximum length of log data that APPC can place in the buffer (that is, the buffer size). The range is from 0 through 65535. Note that a length of zero here indicates that any previous **RECEIVE_LOG_DATA** verb should be cancelled.

log_dptr
Supplied parameter. Specifies the address of the buffer that APPC will use to store the log data.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

REQUEST_TO_SEND

The **REQUEST_TO_SEND** verb notifies the partner TP that the local TP wants to send data.

For the Windows version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **REQUEST_TO_SEND** verb.

```
struct request_to_send {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_REQUEST_TO_SEND.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_R_T_S_BAD_STATE

Secondary return code; the conversation is not in an allowed state when the TP issued this verb.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.

- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any of the following states when the TP issues this verb:

CONFIRM

PENDING_POST (OS/2)

RECEIVE

There is no state change.

The request-to-send notification is received by the partner program through the **rts_rcvd** parameter of the following verbs:

- CONFIRM
- RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems)
- RECEIVE_AND_WAIT
- RECEIVE_IMMEDIATE
- SEND_DATA
- SEND_ERROR

It is also indicated by a **primary_rc** of AP_OK on TEST_RTS.

Request-to-send notification is sent to the partner TP immediately; APPC does not wait until the send buffer fills up or is flushed. Consequently, the request-to-send notification may arrive out of sequence. For example, if the local TP is in SEND state and issues PREPARE_TO_RECEIVE followed by **REQUEST_TO_SEND**, the partner TP, in RECEIVE state, may receive the request-to-send notification before it receives the send notification. For this reason, request-to-send can be reported to a TP through a receive verb.

In response to this request, the partner TP can change the conversation to:

- RECEIVE state by issuing **PREPARE_TO_RECEIVE** or **RECEIVE_AND_WAIT**.
- PENDING_POST state by issuing **RECEIVE_AND_POST**.

The partner TP can also ignore the request-to-send.

The conversation state changes to SEND for the local TP when the local TP receives one of the following values through the **what_rcvd** parameter of a subsequent receive verb:

- AP_CONFIRM_SEND and replies with CONFIRMED
- AP_DATA_COMPLETE_CONFIRM_SEND and replies with **CONFIRMED**
- AP_DATA_CONFIRM_SEND and replies with **CONFIRMED**
- AP_SEND

The receive verbs are RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems), RECEIVE_IMMEDIATE, and RECEIVE_AND_WAIT.

SEND_CONVERSATION

The **SEND_CONVERSATION** verb allocates a session between the local LU and partner LU, sends data on the session, and then deallocates the session.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **SEND_CONVERSATION** verb.

```
struct send_conversation {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3[8];
    unsigned char       rtn_ctl;
    unsigned char       reserv4;
    unsigned long       conv_group_id;
    unsigned long       sense_data;
    unsigned char       plu_alias[8];
    unsigned char       mode_name[8];
    unsigned char       tp_name[64];
    unsigned char       security;
    unsigned char       reserv6[11];
    unsigned char       pwd[10];
    unsigned char       user_id[10];
    unsigned short      pip_dlen;
    unsigned char FAR * pip_dptr;
    unsigned char       reserv6;
    unsigned char       fqplu_name[17];
    unsigned char       reserv7[8];
    unsigned short      dlen;
    unsigned char FAR * dptr;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_SEND_CONVERSATION.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by ALLOCATE in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

rtn_ctl
Supplied parameter. Specifies how APPC should select a session to allocate for the conversation and when the local LU should return control to the local TP. The allowed values are:

- AP_IMMEDIATE specifies that the LU allocates a contention-winner session, if one is immediately available, and returns

control to the TP.

- AP_WHEN_SESSION_ALLOCATED specifies that the LU does not return control to the TP until it allocates a session or encounters one of the errors described in Return Codes in this topic. If the session limit is zero, the LU returns control immediately. Note that if a session is not available, the TP waits for one.
- AP_WHEN_SESSION_FREE specifies that the LU allocates a contention-winner or contention-loser session, if one is available or able to be activated, and returns control to the TP. If an error occurs (as described in Return Codes in this topic) the call will return immediately with the error in the **primary_rc** and **secondary_rc** fields.
- AP_WHEN_CONWINNER_ALLOC specifies that the LU does not return control until it allocates a contention-winner session or encounters one of the errors described in Return Codes in this topic. If the session limit is zero, the LU returns control immediately. Note that if a session is not available, the TP waits for one.
- AP_WHEN_CONV_GROUP_ALLOC specifies that the LU does not return control to the TP until it allocates the session specified by **conv_group_id** or encounters one of the errors described in Return Codes in this topic. If the session is not available, the TP waits for it to become free.

conv_group_id

Supplied/returned parameter. Used as a supplied parameter when **rtn_ctl** is WHEN_CONV_GROUP_ALLOC to specify the identity of the conversation group from which the session should be allocated. When **rtn_ctl** specifies a different value, and the **primary_rc** is AP_OK, this is a returned value. The purpose of this parameter is to provide a TP with the assurance that the same session will be reallocated and therefore the conversations conducted over the session will occur in the same sequence that they were initiated.

sense_data

Returned parameter. If the primary and secondary return codes indicate an allocation error (retry or no-retry), an SNA-defined sense code is returned.

plu_alias

Supplied parameter. Specifies the alias by which the partner LU is known to the local TP. This parameter must match the name of a partner LU established during configuration. The parameter is an 8-byte, type G ASCII character set that includes:

- Uppercase letters
- Numerals 0 to 9
- Spaces
- Special characters $, #, %, and @

If the value of this parameter is fewer than eight bytes, pad it on the right with ASCII spaces (0x20).

mode_name

Supplied parameter. Specifies the name of a set of networking characteristics defined during configuration. This parameter must match the name of a mode associated with the partner LU during configuration.

The parameter is an 8-byte EBCDIC character string. It can consist of characters from the type A EBCDIC character set, including all EBCDIC spaces. These characters are:

- Uppercase letters
- Numerals 0 to 9
- Special characters $, #, and @

The first character in the string must be an uppercase letter or special character.

Using the name SNASVCMG (a reserved mode name used internally by APPC) in a basic conversation is not recommended.

tp_name

Supplied parameter. Specifies the name of the invoked TP. The value of **tp_name** specified by ALLOCATE in the invoking TP must match the value of **tp_name** specified by RECEIVE_ALLOCATE in the invoked TP.

The parameter is a 64-byte, case-sensitive, EBCDIC character string. This parameter can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the TP name is fewer than 64 bytes, use EBCDIC spaces (0x40) to pad it on the right.

The SNA convention for naming a service TP is up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The other characters are from the EBCDIC AE character set.

security
Supplied parameter. Specifies the information the partner LU requires in order to validate access to the invoked TP.

- AP_NONE specifies that the invoked TP uses no conversation security.
- AP_PGM specifies that the invoked TP uses conversation security and requires a user identifier and password. Use **user_id** and **pwd** to supply this information.
- AP_SAME specifies that the invoked TP, invoked with a valid user identifier and password, in turn invokes another TP.

For example, assume that TP A invokes TP B with a valid user identifier and password, and TP B in turn invokes TP C. If TP B specifies the value AP_SAME, APPC will send the LU for TP C the user identifier from TP A and an already-verified indicator. This indicator indicates to TP C not to require the password (if TP C is configured to accept an already-verified indicator).

pwd
Supplied parameter. Specifies the password associated with **user_id**. This parameter is required only if the security parameter is set to AP_PGM and must match the password for **user_id** that was established during configuration.

This parameter is a 10-byte, case-sensitive, EBCDIC character string. It can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the password is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

user_id
Supplied parameter. Specifies the user identifier required to access the partner TP. This parameter is required only if the security parameter is set to AP_PGM and must match one of the user identifiers configured for the partner TP.

The parameter can consist of characters from the type AE EBCDIC character set. These characters are:

- Uppercase and lowercase letters
- Numerals 0 to 9
- Special characters $, #, @, and period (.)

If the user identifier is fewer than 10 bytes, use EBCDIC spaces (0x40) to pad it on the right.

pip_dlen
Supplied parameter. Specifies the length of the PIP to be passed to the partner TP. The range for this parameter is from 0 through 32767.

pip_dptr
Supplied parameter. Specifies the address of the buffer containing PIP data. Use this parameter only if **pip_dlen** is greater than zero.

PIP data can consist of initialization parameters or environmental setup information required by a partner TP or remote operating system. The PIP data must follow the GDS format. For more information, see your IBM SNA manual(s).

For the Microsoft® Windows NT® and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area.

For the OS/2 operating system, use a shared, unnamed segment for the data buffer. To allocate the segment, issue the function call **DosAllocSeg** with the shared indicator equal to 1. The data buffer must be entirely within the data segment.

fqplu_name
Supplied parameter. Specifies the fully qualified name of the local LU. This parameter must match the fully qualified name of the local LU defined in the remote node. The parameter is made up of two type A EBCDIC character strings (each of up to eight characters), which are the network name (NETID) and the LU name of the partner LU. The names are separated by an EBCDIC period (.). The NETID can be omitted, and if this is the case, the period should also be omitted.

This name must be provided if no **plu_alias** is provided.

Type A EBCDIC characters contain:

- Uppercase letters
- Numerals 0 to 9
- Special characters $, #, and @

If the value of this parameter is fewer than 17 bytes, pad it on the right with EBCDIC spaces (0x40).

dlen

Supplied parameter. Specifies the number of bytes of data to be put in the local LU's send buffer. The range for this parameter is from 0 through 65535.

dptr

Supplied parameter. Specifies the address of the buffer containing the data to be put in the local LU's send buffer.

For the Windows NT and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, use a shared, unnamed segment for the data buffer. To allocate the segment, issue the function call **DosAllocSeg** with the shared indicator equal to 1. The data buffer must be entirely within the data segment.

Return Codes

AP_OK

Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL

Primary return code; the supplied parameter **rtn_ctl** specified immediate return of the control to the TP (AP_IMMEDIATE), and the local LU did not have an available contention-winner session.

AP_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

AP_BAD_RETURN_CONTROL

Secondary return code; the value specified for **rtn_ctl** was invalid.

AP_BAD_SECURITY

Secondary return code; the value specified for **security** was invalid.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_PIP_LEN_INCORRECT

Secondary return code; the value of **pip_dlen** was greater than 32767.

AP_UNKNOWN_PARTNER_MODE

Secondary return code; the value specified for **mode_name** was invalid.

AP_BAD_PARTNER_LU_ALIAS

Secondary return code; APPC did not recognize the supplied **partner_lu_alias**.

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with ALLOCATE, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with TP_STARTED is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

This verb is issued by the invoking TP to conduct an entire conversation with the remote TP. If the remote TP rejects either the conversation initiation or the data, the invoking TP will not receive notification of the rejection.

The conversation state is RESET when the TP issues this verb. There is no state change.

Several parameters of **SEND_CONVERSATION** are EBCDIC or ASCII strings. A TP can use the CSV CONVERT to translate a string from one character set to the other.

Normally, the value of **mode_name** must match the name of a mode configured for the invoked TP's node and associated during configuration with the partner LU. If one of the modes associated with the partner LU on the invoked TP's node is an implicit mode, the session established between the two LUs will be of the implicit mode when no mode name associated with the partner LU matches the value of **mode_name**.

SEND_DATA

The **SEND_DATA** verb places data in the local LU's send buffer for transmission to the partner TP.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **SEND_DATA** verb.

```
struct send_data {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       rts_rcvd;
    unsigned char       data_type;
    unsigned short int  dlen;
    unsigned char FAR * dptr ;
    unsigned char       type;
    unsigned char       reserv4;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_SEND_DATA.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier.

The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter. Provides the request-to-send-received indicator.

- AP_YES indicates that the partner TP has issued REQUEST_TO_SEND, which requests that the local TP change the conversation to RECEIVE state. To change to RECEIVE state, the local TP can use PREPARE_TO_RECEIVE, RECEIVE_AND_WAIT, or RECEIVE_AND_POST (Windows NT, Windows 95, and OS/2 operating systems).
- AP_NO indicates that the partner TP has not issued **REQUEST_TO_SEND**.

data_type
Supplied parameter. Specifies the type of data to be sent if Sync Point is supported. Valid parameters are:

AP_APPLICATION

AP_USER_CONTROL_DATA

AP_PS_HEADER

dlen
Supplied parameter. Specifies the number of bytes of data to be put in the local LU's send buffer. The range is from 0 through 65535.

dptr
Supplied parameter. Specifies the address of the buffer containing the data to be put in the local LU's send buffer.

For the Microsoft® Windows NT® and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For OS/2, the data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The data buffer must fit entirely on the data segment.

type
Supplied parameter. Allows a TP to send data and perform other functions within one API call. For example, you can combine **SEND_DATA** with **type** set to CONFIRM to accomplish the same objective as issuing **SEND_DATA** followed by CONFIRM.

- AP_SEND_DATA_CONFIRM corresponds to **SEND_DATA** followed by **CONFIRM**.
- AP_SEND_DATA_FLUSH corresponds to **SEND_DATA** followed by FLUSH.
- AP_SEND_DATA_DEALLOC_ABEND corresponds to **SEND_DATA** followed by DEALLOCATE with a **dealloc_type** of AP_ABEND_PROG.
- AP_SEND_DATA_DEALLOC_FLUSH corresponds to **SEND_DATA** followed by **DEALLOCATE** with a **dealloc_type** of AP_FLUSH.
- AP_SEND_DATA_DEALLOC_SYNC_LEVEL corresponds to **SEND_DATA** followed by **DEALLOCATE** with a **dealloc_type** of AP_SYNC_LEVEL.
- AP_SEND_DATA_P_TO_R_FLUSH corresponds to **SEND_DATA** followed by PREPARE_TO_RECEIVE with a **ptr_type** of AP_FLUSH.
- AP_SEND_DATA_P_TO_R_SYNC_LEVEL corresponds to **SEND_DATA** followed by **PREPARE_TO_RECEIVE** with a **ptr_type** of AP_SYNC_LEVEL and **locks** set to AP_SHORT.

reserv4
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_LL

Secondary return code; the logical record length field of a logical record contained an invalid value—0x0000, 0x0001, 0x8000, or 0x8001. See About Transaction Programs for information on logical records.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the length specified for the data buffer was longer than the segment allocated to contain the buffer.

AP_SEND_DATA_INVALID_TYPE

Secondary return code; the specified type was not recognized by APPC.

AP_SEND_DATA_CONFIRM_SYNC_NONE

Secondary return code; the **type** CONFIRM is not permitted for a conversation that was allocated with a **sync_level** of NONE.

AP_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

AP_SEND_DATA_NOT_SEND_STATE

Secondary return code; the local TP issued **SEND_DATA**, but the conversation was not in SEND state.

AP_SEND_DATA_NOT_LL_BDY

Secondary return code; the TP started but did not finish sending a logical record. This occurs only when the **type** parameter is one of the following:

AP_SEND_DATA_CONFIRM

AP_SEND_DATA_DEALLOC_FLUSH

AP_SEND_DATA_DEALLOC_SYNC_LEVEL

AP_SEND_DATA_P_TO_R_FLUSH

AP_SEND_DATA_P_TO_R_SYNC_LEVEL

AP_ALLOCATION_ERROR

Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **ALLOCATE**, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with **TP_STARTED** is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. Data sent but not yet received is purged.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP has issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC

Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued a **SEND_ERROR** verb with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

Remarks

The conversation must be in SEND state when the TP issues this verb. State changes, based on **primary_rc**, are summarized in the following table.

primary_rc	**New state**
AP_OK	No change
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

SEND_DATA may wait indefinitely because the partner TP has not issued a receive verb. If this occurs, the send buffer may fill up.

The data collected in the local LU's send buffer is transmitted to the partner LU (and partner TP) when one of the following occurs:

- The send buffer fills up.
- The local TP issues FLUSH, CONFIRM, or DEALLOCATE (or other verb that flushes the LU's send buffer).

SEND_ERROR

The **SEND_ERROR** verb notifies the partner TP that the local TP has encountered an application-level error.

For the Microsoft® Windows® version 3.*x* system, it is recommended that you use WinAsyncAPPC rather than the blocking version of this call.

The following structure describes the verb control block used by the **SEND_ERROR** verb.

```
struct send_error {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       rts_rcvd;
    unsigned char       err_type;
    unsigned char       err_dir;
    unsigned char       reserv3;
    unsigned short      log_dlen;
    unsigned char FAR * log_dptr;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_SEND_ERROR.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP.

The value of this parameter is returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter is returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

rts_rcvd
Returned parameter. Indicates whether the partner TP issued REQUEST_TO_SEND. Possible values include:

- AP_YES indicates that the partner TP has issued **REQUEST_TO_SEND**, which requests that the local TP change the conversation to RECEIVE state. To change to RECEIVE state, the local TP can use PREPARE_TO_RECEIVE, RECEIVE_AND_WAIT, or RECEIVE_AND_POST.
- AP_NO indicates that the partner TP has not issued **REQUEST_TO_SEND**.

err_type
Supplied parameter. Indicates the type of the error being reported — application program or service program.

AP_PROG indicates that the error is to be reported to an end-user application program. This value causes APPC to send one of the following return codes to the partner TP:

AP_PROG_ERROR_NO_TRUNC

AP_PROG_ERROR_PURGING

AP_PROG_ERROR_TRUNC

AP_SVC indicates that the error is to be reported to a service program. This value causes APPC to send one of the following return codes to the partner TP:

AP_SVC_ERROR_NO_TRUNC

AP_SVC_ERROR_PURGING

AP_SVC_ERROR_TRUNC

err_dir
Supplied parameter. Indicates whether the error is with data just received or with data that is about to be sent. Use this parameter only when the conversation is in SEND_PENDING state. The parameter is ignored otherwise. The following are allowed values:

- AP_RCV_DIR_ERROR indicates that the TP issued **SEND_ERROR** after detecting an error associated with the data just received.
- AP_SEND_DIR_ERROR indicates that the TP issued **SEND_ERROR** after detecting an error associated with data it was going to send. For example, the TP encountered an error while reading data from the disk drive.

reserv3
A reserved field.

log_dlen
Supplied parameter for basic conversations; specifies the number of bytes of data to be sent to the error log file. The range is from 0 through 32767.

A length of zero indicates that there is no error log data.

log_dptr
Supplied parameter for basic conversations; specifies the address of the data buffer containing error information. The data is sent to the local error log and to the partner LU.

This parameter is used by **SEND_ERROR** if **log_dlen** is greater than zero.

For the Microsoft® Windows NT® and Windows 95 operating systems and the Windows graphical environment, the data buffer can reside in a static data area or in a globally allocated area. The data buffer must fit entirely within this area.

For the OS/2 operating system, the log data buffer must reside on an unnamed, shared segment, which is allocated by the function **DosAllocSeg** with **Flags** equal to 1. The log data buffer must fit entirely on the segment.

The TP must format the error data as a GDS error log variable. For more information, see your IBM SNA manual(s).

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_BAD_ERROR_DIRECTION

Secondary return code; the specified **err_dir** was not recognized by APPC.

AP_INVALID_DATA_SEGMENT

Secondary return code; the error data for the log file was longer than the segment allocated to contain the error data, or the address of the error data buffer was wrong.

AP_SEND_ERROR_BAD_TYPE

Secondary return code; the value of **err_type** was invalid.

AP_SEND_ERROR_LOG_LL_WRONG

Secondary return code; the LL field of the error log GDS variable did not match the actual length of the data.

The following return codes can be generated when **SEND_ERROR** is issued in any allowed state:

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **ALLOCATE**, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with **TP_STARTED** is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONV_FAILURE_NO_RETRY
Primary return code; the conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

AP_CONV_FAILURE_RETRY
Primary return code; the conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the system administrator should examine the error log to determine the cause of the error.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This may occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

The following return codes can be generated only if **SEND_ERROR** is issued in SEND state:

AP_ALLOCATION_ERROR
Primary return code; APPC has failed to allocate a conversation. The conversation state is set to RESET.

This code may be returned through a verb issued after ALLOCATE.

AP_ALLOCATION_FAILURE_NO_RETRY

Secondary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

AP_ALLOCATION_FAILURE_RETRY

Secondary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

AP_CONVERSATION_TYPE_MISMATCH

Secondary return code; the partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

AP_PIP_NOT_ALLOWED

Secondary return code; the allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

AP_PIP_NOT_SPECIFIED_CORRECTLY

Secondary return code; the partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

AP_SECURITY_NOT_VALID

Secondary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

AP_SYNC_LEVEL_NOT_SUPPORTED

Secondary return code; the partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

AP_TP_NAME_NOT_RECOGNIZED

Secondary return code; the partner LU does not recognize the TP name specified in the allocation request.

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

AP_TRANS_PGM_NOT_AVAIL_RETRY

Secondary return code; the remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

AP_PROG_ERROR_PURGING
Primary return code; while in RECEIVE, PENDING, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the partner TP issued **SEND_ERROR** with **err_type** set to AP_PROG. Data sent but not yet received is purged.

The following return codes can be generated only if **SEND_ERROR** is issued in SEND state:

AP_DEALLOC_ABEND_PROG
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner TP has issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP has encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

AP_DEALLOC_ABEND_SVC
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_SVC.

AP_DEALLOC_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner TP issued **DEALLOCATE** with **dealloc_type** set to AP_ABEND_TIMER.

AP_SVC_ERROR_PURGING
Primary return code; the partner TP (or partner LU) issued **SEND_ERROR** with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Windows NT, Windows 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

The following return code can be generated only if **SEND_ERROR** is issued in RECEIVE state:

AP_DEALLOC_NORMAL
Primary return code; this return code does not indicate an error.

The partner TP issued DEALLOCATE with **dealloc_type** set to one of the following:

- AP_FLUSH
- AP_SYNC_LEVEL with the synchronization level of the conversation specified as AP_NONE

Remarks

The conversation can be in any state except RESET when the TP issues this verb. The conversation state must be SEND_PENDING if **err_dir** is used.

The local TP sends the error notification immediately to the partner TP; it does not hold the information in the local LU's send buffer.

Upon successful execution of this verb, the conversation is in SEND state for the local TP and in RECEIVE state for the partner TP.

The new state is determined by **primary_rc**. Possible state changes are summarized in the following table.

primary_rc	New state
AP_OK	SEND
AP_ALLOCATION_ERROR	RESET
AP_CONV_FAILURE_RETRY	RESET
AP_CONV_FAILURE_NO_RETRY	RESET
AP_DEALLOC_ABEND	RESET
AP_DEALLOC_ABEND_PROG	RESET
AP_DEALLOC_ABEND_SVC	RESET
AP_DEALLOC_ABEND_TIMER	RESET
AP_DEALLOC_NORMAL	RESET
AP_PROG_ERROR_PURGING	RECEIVE
AP_SVC_ERROR_PURGING	RECEIVE

If the conversation is in RECEIVE state when the TP issues **SEND_ERROR**, incoming data is purged by APPC. This data includes:

- Data sent by SEND_DATA.
- Return code indicators.
- Confirmation requests.
- Deallocation requests.

APPC does not purge an incoming request-to-send indicator. APPC replaces purged incoming return code indicators with other return codes. The primary return code AP_OK replaces the following purged return code indicators:

AP_PROG_ERROR_NO_TRUNC

AP_PROG_ERROR_PURGING

AP_PROG_ERROR_TRUNC

AP_SVC_ERROR_NO_TRUNC

AP_SVC_ERROR_PURGING

AP_SVC_ERROR_TRUNC

The primary return code AP_DEALLOC_NORMAL replaces the following purged return code indicators:

AP_ALLOCATION_ERROR

AP_ALLOCATION_FAILURE_NO_RETRY

AP_ALLOCATION_FAILURE_RETRY

AP_CONVERSATION_TYPE_MISMATCH

AP_DEALLOC_ABEND

AP_DEALLOC_ABEND_PROG

AP_DEALLOC_ABEND_SVC

AP_DEALLOC_ABEND_TIMER

AP_PIP_NOT_ALLOWED

AP_PIP_NOT_SPECIFIED_CORRECTLY

AP_SECURITY_NOT_VALID

AP_SYNC_LEVEL_NOT_SUPPORTED

AP_TP_NAME_NOT_RECOGNIZED

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY

AP_TRANS_PGM_NOT_AVAIL_RETRY

When the conversation is in SEND_PENDING state, APPC reports the following return codes to the partner TP based on the value in **err_dir**:

AP_PROG_ERROR_PURGING
The local TP issued **SEND_ERROR** with RECEIVE as the **err_dir**.
AP_PROG_ERROR_NO_TRUNC
The local TP issued **SEND_ERROR** with SEND as the **err_dir**.
AP_SVC_ERROR_PURGING
The local TP issued **SEND_ERROR** with RECEIVE as the **err_dir.**
AP_SVC_ERROR_NO_TRUNC
The local TP issued **SEND_ERROR** with SEND as the **err_dir**.

TEST_RTS

The **TEST_RTS** verb determines whether a request-to-send notification has been received from the partner TP.

The following structure describes the verb control block used by the **TEST_RTS** verb.

```
struct test_rts {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_TEST_RTS.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter was returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

Return Codes

AP_OK
Primary return code; the verb executed successfully.

AP_UNSUCCESSFUL
Primary return code; request-to-send notification has not been received.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID
Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **ALLOCATE**, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with **TP_STARTED** is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED
Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.
AP_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
AP_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
AP_CONV_BUSY
Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.
AP_THREAD_BLOCKING
Primary return code; the calling thread is already in a blocking call.
AP_UNEXPECTED_DOS_ERROR
Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Remarks

The conversation can be in any state except RESET when the TP issues this verb.

There is no state change.

TEST_RTS_AND_POST

The **TEST_RTS_AND_POST** verb allows an application, typically a 5250 emulator, to request asynchronous notification when a partner TP requests send direction. It is not supported on Microsoft® MS-DOS® platforms.

The following structure describes the verb control block used by the **TEST_RTS_AND_POST** verb.

```
struct test_rts {
    unsigned short      opcode;
    unsigned char       opext;
    unsigned char       reserv2;
    unsigned short      primary_rc;
    unsigned long       secondary_rc;
    unsigned char       tp_id[8];
    unsigned long       conv_id;
    unsigned char       reserv3;
    unsigned long       handle;
};
```

Members

opcode
Supplied parameter. Specifies the verb operation code, AP_B_TEST_RTS_AND_POST.

opext
Supplied parameter. Specifies the verb operation extension, AP_BASIC_CONVERSATION.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

tp_id
Supplied parameter. Identifies the local TP. The value of this parameter was returned by TP_STARTED in the invoking TP or by RECEIVE_ALLOCATE in the invoked TP.

conv_id
Supplied parameter. Provides the conversation identifier. The value of this parameter was returned by ALLOCATE in the invoking TP or by **RECEIVE_ALLOCATE** in the invoked TP.

reserv3
A reserved field.

handle
Supplied parameter. On Microsoft® Windows NT® and Microsoft® Windows® 95 this field provides the event handle to set. On Windows 3.x, this field provides the Windows handle to receive the completion message. On OS/2, this field provides the address of the semaphore APPC is to clear when the asynchronous operation is finished.

Return Codes from Initial Verb

AP_OK
Primary return code; the verb executed successfully. Note particularly that a return code of AP_OK from the initial verb does not indicate that **REQUEST_TO_SEND** verb received from the partner TP. It simply indicates that the facility to receive asynchronous notification has been registered.

AP_UNSUCCESSFUL
Primary return code; request-to-send notification has not been received.

AP_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

AP_BAD_CONV_ID

Secondary return code; the value of **conv_id** did not match a conversation identifier assigned by APPC.

AP_BAD_TP_ID

Secondary return code; the value of **tp_id** did not match a TP identifier assigned by APPC.

AP_INVALID_SEMAPHORE_HANDLE

Secondary return code, the value of **handle** was invalid.

AP_COMM_SUBSYSTEM_ABENDED

Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

AP_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

When this return code is used with **ALLOCATE**, it may indicate that no communications subsystem could be found to support the local LU. (For example, the local LU alias specified with **TP_STARTED** is incorrect or has not been configured.) Note that if **lu_alias** or **mode_name** is fewer than eight characters, you must ensure that these fields are filled with spaces to the right. This error is returned if these parameters are not filled with spaces, since there is no node available that can satisfy the **ALLOCATE** request.

When **ALLOCATE** produces this return code for a Microsoft® Host Integration Server 2000 Client system configured with multiple nodes, there are two secondary return codes as follows:

0xF0000001

Secondary return code; no nodes have been started.

0xF0000002

Secondary return code; at least one node has been started, but the local LU (when **TP_STARTED** is issued) is not configured on any active nodes. The problem could be either of the following:

- The node with the local LU is not started.
- The local LU is not configured.

AP_CONVERSATION_TYPE_MIXED

Primary return code; the TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

AP_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

AP_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

AP_CONV_BUSY

Primary return code; there can only be one outstanding conversation verb at a time on any conversation. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **conv_id**.

AP_THREAD_BLOCKING

Primary return code; the calling thread is already in a blocking call.

AP_UNEXPECTED_DOS_ERROR

Primary return code; the operating system has returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

Return Codes from Asynchronous Completion

AP_OK

Primary return code; the request-to-send notification has been received from the partner TP.

AP_CANCELLED

The outstanding **TEST_RTS_AND_POST** verb has been terminated. This will occur if the underlying conversation has been deallocated or an AP_TP_ENDED has been issued.

Note that as with **RECEIVE_AND_POST**, the TP is still responsible for correctly terminating the conversation and possibly terminating the TP. Issuing another verb, such as **RECEIVE_IMMEDIATE,** at this point will indicate the reason for the conversation failure.

The conversation can be in any state except RESET when the TP issues this verb. There is no state change.

A common feature of many APPC applications, such as 5250 emulators, is a requirement to detect a partner's request to send. Currently, this can be done by polling the APPC interface to detect the partner's request. For example, an application can occasionally issue one of the following verbs:

- **TEST_RTS**

Remarks

- **RECEIVE_IMMEDIATE** and check the **rts_rcvd** field
- **SEND_DATA** of zero bytes, again checking the **rts_rcvd** field.

Some of the problems associated with this polling approach are:

- The application must continually interrupt its main work to poll APPC.
- The partner's request is not detected as soon as it becomes available.
- These approaches are processor-intensive.

The **TEST_RTS_AND_POST** verb allows an application running on Windows NT, Windows 95, Windows 3.*x*, or OS/2, typically a 5250 emulator, to request asynchronous notification when the partner TP requests send direction.

An APPC application typically issues the **TEST_RTS_AND_POST** verb while in SEND state and then continues with its main processing. A request for send direction from the partner TP is indicated asynchronously to the application. After dealing with the partner's request, the application typically returns to SEND state, reissues **TEST_RTS_AND_POST**, and continues.

The **TEST_RTS_AND_POST** verb completes synchronously and the return code AP_OK indicates that a request for asynchronous notification has been registered. It is important to emphasize that this does not indicate that request-to-send was received from the partner TP.

When the partner's request to send is received, the asynchronous event completion occurs. It is important to note that this may be before the completion of the local TP's original **TEST_RTS_AND_POST** verb. This will be the case if the partner's request to send was received before the local TP's **TEST_RTS_AND_POST** verb was issued, or while the local TP's **TEST_RTS_AND_POST** verb was being processed.

APPC Extensions for the Windows Environment

This section describes API extensions to Windows APPC that allow asynchronous communication. Asynchronous communication occurs when a function returns before the request completes. The application is notified later when the request is completed.

Under Microsoft® Windows®3.x, one method is are available for asynchronous communication using the APPC API:

- Message posting using window handles.

Under Microsoft Windows 98, and Windows® 95, two methods are available for asynchronous communication using the APPC API:

- Message posting using window handles.
- Waiting on Win32® events.

Under Microsoft Windows 2000 and Windows NT®, three methods are available for asynchronous communication using the APPC API:

- Message posting using window handles.
- Waiting on Win32 events.
- Using Win32 IO completion ports.

The first method uses messages posted to a window handle to notify an application of verb completion. This method using window handles and messages is supported on Microsoft Windows 3.x with the exception of real mode Windows 3.0. There is one such window for each APPC application, independent of the number of conversations. Each APPC conversation can have one asynchronous verb outstanding at any time. When a verb completes, the posting to the window takes as parameters the asynchronous task handle returned by the original call and a pointer to the verb control block which has completed, containing the return codes of the verb.

The extensions using window handles and message posting described in this section (WinAsyncAPPC) have been designed for all implementations and versions of Microsoft Windows from version 3.0 through the latest versions of Windows 2000, Windows NT, Windows 98, and Windows 95. They provide compatibility for Windows programming and optimum application performance in the 16-bit Windows operating environment.

A second method using Win32 events for notification is supported. The extensions using Win32 events described in this section (WinAsyncAPPCEx) operate only on Windows 2000, Windows NT, Windows 98, and Windows 95. and offer optimum application performance in the 32-bit Windows operating environment. If an event has been registered with the conversation, then an application can call the Win32 **WaitForSingleObject** or **WaitForMultipleObjects** function to wait to be notified of the completion of the verb.

A third method using Win32 I/O completion ports for notification is supported on Windows 2000 and Windows NT. The extensions using I/O completion ports described in this section (WinAsyncAPPCIOCP) operate only on Windows 2000 and Windows NT, and offer optimum application performance in the 32-bit Windows operating environment. If an I/O completion port has been created using **CreateIoCompletionPort**, then an application can call the Win32 **GetQueuedCompletionStatus** function to wait to be notified of the completion of the verb.

Windows APPC allows multithreaded Windows-based processes. A process contains one or more threads of execution. The 16-bit Windows environment is not multithreaded. In this instance, a task corresponds to a process with a single thread. All references to threads in this document refer to actual threads in multithreaded Windows environments.

This section provides, for each extension, a definition of the function, syntax, returns, and remarks for using the function.

WinAsyncAPPC

The **WinAsyncAPPC** function provides an asynchronous entry point for all of the APPC verbs. Use this function instead of the blocking versions of the verbs if you run your application and want to use message posting using Windows handles for asynchronous verb completion. This function is supported on Windows 3.x

```
HANDLE WINAPI WinAsyncAPPC(
    HANDLE hWnd,
    long lpVcb
);
```

Parameters

hWnd
A window handle that will be used for message posting to notify an application when an APPC verb completes.
lpVcb
Pointer to the verb control block

Return Values

The return value specifies whether the asynchronous request was successful. If the function was successful, the return value is an asynchronous task handle. If the function was not successful, a zero is returned.

When this function returns with a successful value, this does not indicate that the APPC call will ultimately return successfully. It only indicates that it was possible for the APPC library to attempt the APPC call asynchronously using message posting for notification.

Remarks

For an example of how to use this verb in TPs, see the send and receive sample TP (SENDRECV.C located in the APPC folder) included in the SDK.

APPC verbs used in basic conversations that can block are as follows:

- ALLOCATE
- CONFIRM
- CONFIRMED
- DEALLOCATE
- FLUSH
- PREPARE_TO_RECEIVE
- RECEIVE_ALLOCATE
- RECEIVE_AND_WAIT
- REQUEST_TO_SEND
- SEND_CONVERSATION
- SEND_DATA
- SEND_ERROR
- TP_ENDED
- TP_STARTED

APPC verbs used in mapped conversations that can block are as follows:

- MC_ALLOCATE
- MC_CONFIRM
- MC_CONFIRMED
- MC_DEALLOCATE
- MC_FLUSH
- MC_PREPARE_TO_RECEIVE
- MC_RECEIVE_AND_WAIT
- MC_REQUEST_TO_SEND
- MC_SEND_CONVERSATION

- MC_SEND_DATA
- MC_SEND_ERROR

When using the synchronous or asynchronous versions of a verb, an application can only have one outstanding function in progress on a conversation at a time. An attempt to initiate a second function results in the error code AP_CONV_BUSY.

> The exceptions to the preceding paragraph are RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT. To allow full use of the asynchronous support, asynchronously issued **RECEIVE_AND_WAIT** and **MC_RECEIVE_AND_WAIT** verbs have been altered to act like the **RECEIVE_AND_POST** and **MC_RECEIVE_AND_POST** verbs. Specifically, while an asynchronous version of one of these verbs is outstanding, the following verbs can be issued on the same conversation:

- DEALLOCATE (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)
- GET_ATTRIBUTES or MC_GET_ATTRIBUTES
- GET_TYPE
- REQUEST_TO_SEND or MC_REQUEST_TO_SEND
- SEND_ERROR or MC_SEND_ERROR
- TEST_RTS or MC_TEST_RTS
- TP_ENDED

This allows an application, in particular, a 5250 emulator, to use an asynchronous **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_POST**, **MC_RECEIVE_AND_POST**, **RECEIVE_AND_WAIT**, or **MC_RECEIVE_AND_WAIT** is outstanding, it can still use **SEND_ERROR** or **MC_SEND_ERROR** and **REQUEST_TO_SEND** or **MC_REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support.

When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinAsyncAPPC" as the input string. The *wParam* argument contains the asynchronous task handle returned by the original function call. The *lParam* argument contains the original VCB pointer and can be dereferenced to determine the final return code.

As part of the Windows APPC definition, WinAPPCCancelAsyncRequest allows an application to cancel any asynchronous APPC action; but terminates the related conversation or TP as appropriate. Any outstanding operations return with AP_CANCELED as the return code.

If the function returns successfully, a **WinAsyncAPPC** message is posted to the application when the operation completes or the conversation is canceled.

WinAsyncAPPCEx

The **WinAsyncAPPCEx** function provides an asynchronous entry point for all of the APPC verbs. Use this function instead of the blocking versions of the verbs to allow multiple sessions to be handled on the same thread using events. This verb is only supported on Microsoft® Windows 2000, Windows NT®, Windows® 98, and Windows® 95, and uses Win32® events.

```
HANDLE WINAPI WinAsyncAPPCEx(
    HANDLE event_handle,
    long lpVcb
);
```

Parameters

event_handle
Handle used for event notification using Win32 events.
lpVcb
Pointer to the verb control block

Return Values

The return value specifies whether the asynchronous resolution request was successful. If the function was successful, the return value is an asynchronous task handle. If the function was not successful, a zero is returned.

When this function returns with a successful value, this does not indicate that the APPC call will ultimately return successfully. It only indicates that it was possible for the APPC library to attempt the APPC call asynchronously using events for notification.

Remarks

This function is intended for use with **WaitForSingleObject** or **WaitForMultipleObjects** in the Win32 API. These functions are described in the "Reference" section of the Microsoft® Platform SDK documentation.

For an example of how to use this verb in multithreaded TPs, see the multithreaded send and receive sample TPs (MRCV.C MSEND.C, and MSENDRCV.C located in the MSENDRCV folder) included in the SDK.

APPC verbs used in basic conversations that can block are as follows:

- ALLOCATE
- CONFIRM
- CONFIRMED
- DEALLOCATE
- FLUSH
- PREPARE_TO_RECEIVE
- RECEIVE_ALLOCATE
- RECEIVE_AND_WAIT
- REQUEST_TO_SEND
- SEND_CONVERSATION
- SEND_DATA
- SEND_ERROR
- TP_ENDED
- TP_STARTED

APPC verbs used in mapped conversations that can block are as follows:

- MC_ALLOCATE
- MC_CONFIRM
- MC_CONFIRMED
- MC_DEALLOCATE
- MC_FLUSH
- MC_PREPARE_TO_RECEIVE
- MC_RECEIVE_AND_WAIT
- MC_REQUEST_TO_SEND

- MC_SEND_CONVERSATION
- MC_SEND_DATA
- MC_SEND_ERROR
- RECEIVE_ALLOCATE
- TP_ENDED
- TP_STARTED

When using the synchronous or asynchronous versions of a verb, an application can only have one outstanding function in progress on a conversation at a time. An attempt to initiate a second function results in the error code AP_CONV_BUSY.

> The exceptions to the preceding paragraph are RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT. To allow full use of the asynchronous support, asynchronously issued **RECEIVE_AND_WAIT** and **MC_RECEIVE_AND_WAIT** verbs have been altered to act like the **RECEIVE_AND_POST** and **MC_RECEIVE_AND_POST** verbs. Specifically, while an asynchronous version of one of these verbs is outstanding, the following verbs can be issued on the same conversation:

- DEALLOCATE (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)
- GET_ATTRIBUTES or MC_GET_ATTRIBUTES
- GET_TYPE
- REQUEST_TO_SEND or MC_REQUEST_TO_SEND
- SEND_ERROR or MC_SEND_ERROR
- TEST_RTS or MC_TEST_RTS
- TP_ENDED

This allows an application, in particular, a server application, to use an asynchronous **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_POST**, **MC_RECEIVE_AND_POST**, **RECEIVE_AND_WAIT**, or **MC_RECEIVE_AND_WAIT** is outstanding, it can still use **SEND_ERROR** or **MC_SEND_ERROR** and **REQUEST_TO_SEND** or **MC_REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support, and in particular, for support of multiple conversations on the same thread.

When the asynchronous operation is complete, the application is notified through the signaling of the event. Upon signaling of the event, examine the APPC primary return code and secondary return code in the verb control block for any error conditions.

WinAsyncAPPCIOCP

The **WinAsyncAPPCIOCP** function provides an asynchronous entry point for all of the APPC verbs. Use this function instead of the blocking versions of the verbs to allow multiple sessions to be handled on the same thread using I/O completion ports. This verb is only supported on Microsoft® Windows 2000 and Windows NT®, and uses Win32® I/O completion ports.

```
HANDLE WINAPI WinAsyncAPPCIOCP(
    APPC_IOCP_INFO *iocp_handle,
    long lpVcb
);
```

Parameters

iocp_handle
A pointer to an APPC_IOCP_INFO structure used for passing I/O completion port information.
lpVcb
Pointer to the verb control block

The APPC_IOCP_INFO structure has the following prototype:

```
typedef struct {
    HANDLE APPC_CompletionPort;
    DWORD APPC_NumberOfBytesTransferred;
    DWORD APPC_CompletionKey;
    LPOVERLAPPED APPC_pOverlapped;
} APPC_IOCP_INFO;
```

APPC_CompletionPort

This supplied parameter is the HANDLE returned by the call to the **CreateIoCompletionPort**.function when the I/O completion port is created. The I/O completion port must be created before calling the **WinAsyncAPPCIOCP** function.
When the verb completes, the APPC Library calls the **PostQueuedCompletionStatus** with the remaining fields in the structure as inputs, and these fields are simply passed through to the **GetQueuedCompletionStatus** function issued by the application.

APPC_NumberOfBytesTransferred

This supplied parameter is ignored. When the APPC verb completes, the APPC Library calls the **PostQueuedCompletionStatus** function with this field as an input, and the value returned for the *dwNumberOfBytesTransferred* is simply passed through to the **GetQueuedCompletionStatus** function issued by the application.

APPC_CompletionKey

This supplied parameter is ignored. When the APPC verb completes, the APPC Library calls the **PostQueuedCompletionStatus** function with this field as an input, and the value returned for the *dwCompletionKey* is simply passed through to the **GetQueuedCompletionStatus** function issued by the application.

APPC_pOverlapped

This supplied parameter is ignored. When the APPC verb completes, the APPC Library calls the **PostQueuedCompletionStatus** function with this field as an input, and the value returned for the *lpOverlapped* is simply passed through to the **GetQueuedCompletionStatus** function issued by the application.

Return Values

The return value specifies whether the asynchronous resolution request was successful. If the function was successful, the return value is an asynchronous task handle. If the function was not successful, a zero is returned.

When this function returns with a successful value, this does not indicate that the APPC call will ultimately return successfully. It only indicates that it was possible for the APPC library to attempt the APPC call asynchronously using an I/O completion port for notification.

Remarks

This function is intended for use with **CreateIoCompletionPort** and **GetQueuedCompletionStatus** in the Win32 API. These functions are described in the "Reference" section of the Microsoft® Platform SDK documentation.

For an example of how to use this verb in multithreaded TPs, see the multithreaded receive sample TP (MRCVIO located in the

SNA\MSENDRCV folder) using I/O completion ports included in the Host Integration Server 2000 SDK.

APPC verbs used in basic conversations that can block are as follows:

- ALLOCATE
- CONFIRM
- CONFIRMED
- DEALLOCATE
- FLUSH
- PREPARE_TO_RECEIVE
- RECEIVE_ALLOCATE
- RECEIVE_AND_WAIT
- REQUEST_TO_SEND
- SEND_CONVERSATION
- SEND_DATA
- SEND_ERROR
- TP_ENDED
- TP_STARTED

APPC verbs used in mapped conversations that can block are as follows:

- MC_ALLOCATE
- MC_CONFIRM
- MC_CONFIRMED
- MC_DEALLOCATE
- MC_FLUSH
- MC_PREPARE_TO_RECEIVE
- MC_RECEIVE_AND_WAIT
- MC_REQUEST_TO_SEND
- MC_SEND_CONVERSATION
- MC_SEND_DATA
- MC_SEND_ERROR
- RECEIVE_ALLOCATE
- TP_ENDED
- TP_STARTED

When using the synchronous or asynchronous versions of a verb, an application can only have one outstanding function in progress on a conversation at a time. An attempt to initiate a second function results in the error code AP_CONV_BUSY.

> The exceptions to the preceding paragraph are RECEIVE_AND_POST, MC_RECEIVE_AND_POST, RECEIVE_AND_WAIT, and MC_RECEIVE_AND_WAIT. To allow full use of the asynchronous support, asynchronously issued **RECEIVE_AND_WAIT** and **MC_RECEIVE_AND_WAIT** verbs have been altered to act like the **RECEIVE_AND_POST** and **MC_RECEIVE_AND_POST** verbs. Specifically, while an asynchronous version of one of these verbs is outstanding, the following verbs can be issued on the same conversation:

- DEALLOCATE (AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER)
- GET_ATTRIBUTES or MC_GET_ATTRIBUTES
- GET_TYPE
- REQUEST_TO_SEND or MC_REQUEST_TO_SEND
- SEND_ERROR or MC_SEND_ERROR
- TEST_RTS or MC_TEST_RTS
- TP_ENDED

This allows an application, in particular, a server application, to use an asynchronous **RECEIVE_AND_WAIT** or **MC_RECEIVE_AND_WAIT** to receive data. While the **RECEIVE_AND_POST**, **MC_RECEIVE_AND_POST**, **RECEIVE_AND_WAIT**, or **MC_RECEIVE_AND_WAIT** is outstanding, it can still use **SEND_ERROR** or **MC_SEND_ERROR** and **REQUEST_TO_SEND** or **MC_REQUEST_TO_SEND**. It is recommended that you use this feature for full asynchronous support, and in particular, for support of multiple conversations on the same thread.

When the asynchronous operation is complete, the application is notified through the **GetQueuedCompletionStatus** function. Upon I/O completion, examine the APPC primary return code and secondary return code in the verb control block for any error conditions.

WinAPPCCancelAsyncRequest

The **WinAPPCCancelAsyncRequest** function cancels an outstanding WinAsyncAPPC-based request.

```
int WINAPI WinAPPCCancelAsyncRequest(
    HANDLE hAsyncTaskID
);
```

Parameters

hAsyncTaskID
Supplied parameter. Specifies the asynchronous task to be canceled.

Return Values

The return value specifies whether the asynchronous request was canceled. If the value is zero, the request was canceled. Otherwise, the value is one of the following:

WAPPCINVALID
An error code indicating that the specified asynchronous task identifier was invalid.
WAPPCALREADY
An error code indicating that the asynchronous routine being canceled has already completed.

Remarks

An asynchronous task previously initiated by issuing one of the **WinAsyncAPPC**, **WinAsyncAPPCEx**, or **WinAsyncAPPCIOCP** functions can be canceled prior to completion by issuing the **WinAPPCCancelAsyncRequest** function, specifying the asynchronous task identifier as returned by the initial function in *hAsyncTaskID*.

If the outstanding verb relates to a conversation (for example, SEND_DATA or RECEIVE_AND_WAIT), the verb is purged and the session is closed. If the verb relates to a TP (for example, RECEIVE_ALLOCATE or TP_STARTED), the TP is ended. In both cases, while the implementation closes conversations and sessions as cleanly as possible, it does not flush send buffers, wait for confirmations, and so on. This call is synchronous, and after the processing described above is complete, a completion message is posted for the canceled verb.

If an attempt to cancel an existing asynchronous **WinAsyncAPPC** routine fails with an error code of WAPPCALREADY, one of two things has occurred. Either the original routine has already completed and the application has dealt with the resulting message, or the original routine has already completed and the resulting message is still waiting in the application window queue.

WinAPPCCancelBlockingCall

The **WinAPPCCancelBlockingCall** function cancels any outstanding blocking operation for its thread. Any outstanding blocked call canceled will cause an error code of WAPPCCANCEL to be generated.

```
BOOL WINAPI WinAPPCCancelBlockingCall(void);
```

Return Values

The return value specifies whether the cancellation request was successful. If the value is zero, the request was canceled. Otherwise, the value is the following:

WAPPCINVALID
An error code indicating that there is no outstanding blocking call.

Remarks

If the outstanding verb relates to a conversation (for example, SEND_DATA or RECEIVE_AND_WAIT), the verb is purged and the session is closed. If the verb relates to a TP (for example, RECEIVE_ALLOCATE or TP_STARTED), the TP is ended. In both cases, while the implementation brings down conversations and sessions as cleanly as possible, it does not flush send buffers, wait for confirmations, and so on. This call is synchronous and after the processing described above is complete, the function is finished.

In Microsoft® Windows 2000, Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95, a multithreaded application can have multiple blocking operations outstanding, but only one per thread. To distinguish between multiple outstanding calls, **WinAPPCCancelBlockingCall** cancels the outstanding operation on the current, or calling, application thread if one exists; otherwise, it fails. By default in Windows 2000, Windows NT, Windows 98, and Windows 95, Windows APPC suspends the calling application thread while an operation is outstanding. As a result, the thread on which the blocking operation was initiated will not regain control (and therefore, will not be able to issue a call to **WinAPPCCancelBlockingCall**) unless a blocking hook is registered for the thread using WinAPPCSetBlockingHook. This condition does not apply to Microsoft® Windows® version 3*.x* since applications only have one effective thread and the default blocking hook is registered by default.

WinAPPCCleanup

The **WinAPPCCleanup** function terminates and deregisters an application from a Windows APPC implementation.

```
BOOL WINAPI WinAPPCCleanup(void);
```

Return Values

The return value specifies whether the deregistration was successful. If the value is nonzero, the application was successfully deregistered. The application was not deregistered if a value of zero is returned.

Remarks

Use **WinAPPCCleanup** to indicate deregistration of a Windows APPC application from a Windows APPC implementation.

Conversations that are still active will be terminated and TPs ended. This function is equivalent to issuing TP_ENDED(HARD) on all TPs owned by the application.

See Also

WinAPPCStartup

WinAPPCIsBlocking

The **WinAPPCIsBlocking** function determines if a thread is executing while waiting for a previous blocking call to finish.

```
BOOL WINAPI WinAPPCIsBlocking(void);
```

Return Values

The return value specifies the outcome of the function. If the value is nonzero, there is an outstanding blocking call awaiting completion. A zero indicates the absence of an outstanding blocking call.

Remarks

Although a call issued on a blocking function appears to an application as though it blocks, the Windows APPC DLL has to relinquish the processor to allow other applications to run. This means that it is possible for the application that issued the blocking call to be re-entered, depending on the message(s) it receives. In this instance, the **WinAPPCIsBlocking** call can be used to determine whether the application task currently has been re-entered while waiting for an outstanding blocking call to finish. Note that Windows APPC prohibits more than one outstanding blocking call per thread.

The Windows APPC DLL prohibits more than one blocking call per thread and returns AP_THREAD_BLOCKING if this occurs.

See Also

WinAPPCSetBlockingHook, WinAPPCUnhookBlockingHook, WinAPPCCancelBlockingCall

WinAPPCStartup

The **WinAPPCStartup** function allows an application to specify the version of Windows APPC required and to retrieve details of the specific Windows APPC implementation. An application must call this function to register itself with a Windows APPC implementation before issuing any further Windows APPC calls.

```
int WINAPI WinAPPCStartup(
    WORD wVersionRequired,
    LPWAPPCDATA lpAPPCData
);

typedef struct {
    WORD wVersion;
    char szDescription[WAPPCDESCRIPTION_LEN+1];
} WAPPCDATA, FAR * LPWAPPCDATA;

where WAPPCDESCRIPTION_LEN is defined as 127
```

Parameters

wVersionRequired
Specifies the version of Windows APPC support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number. The current version of the Windows APPC API is 1.0.

lpAPPCData
Pointer to a returned structure containing a Windows APPC version number and a description of the Windows APPC implementation.

Return Values

The return value specifies whether the application was registered successfully and whether the Windows APPC implementation can support the specified version number. If the value is zero, it was registered successfully and the specified version can be supported. Otherwise, the return value is one of the following:

WAPPCSYSNOTREADY
The underlying network subsystem is not ready for network communication.

WAPPCVERNOTSUPPORTED
The version of Windows APPC support requested is not provided by this particular Windows APPC implementation.

WAPPCINVALID
The Windows APPC version specified by the application is not supported by this DLL.

Remarks

To support future Windows APPC implementations and applications that may have functionality differences from Windows APPC version 1.0, a negotiation takes place in **WinAPPCStartup**. An application passes to **WinAPPCStartup** the Windows APPC version that it can use. If this version is lower than the lowest version supported by the Windows APPC DLL, the DLL cannot support the application and **WinAPPCStartup** fails. If the version is not lower, however, the call succeeds and returns the highest version of Windows APPC supported by the DLL. If this version is lower than the lowest version supported by the application, the application either fails its initialization or attempts to find another Windows APPC DLL on the system.

This negotiation allows both a Windows APPC DLL and a Windows APPC application to support a range of Windows APPC versions. An application can successfully use a DLL if there is any overlap in the versions. The following table illustrates how **WinAPPCStartup** works in conjunction with different application and DLL versions.

Application versions	DLL versions	To WinAPPCStartup	From WinAPPCStartup	Result
1.0	1.0	1.0	1.0	Use 1.0
1.0, 2.0	1.0	2.0	1.0	Use 1.0
1.0	1.0, 2.0	1.0	2.0	Use 1.0
1.0	2.0, 3.0	1.0	WAPPCINVALID	Fail
2.0, 3.0	1.0	3.0	1.0	App Fails
1.0, 2.0, 3.0	1.0, 2.0, 3.0	3.0	3.0	Use 3.0
1.0	1.0	4.0	1.0	App Fails
1.0, 2.0, 3.0, 4.0	1.0, 2.0, 3.0, 4.0	4.0	4.0	Use 4.0

Details of the actual Windows APPC implementation are described in the **WAPPCDATA** structure defined as follows that is

returned by **WinAPPCStartup**:

```
typedef struct tagWAPPCDDATA { WORD wVersion;
char szDescription[WAPPCDESCRIPTION_LEN+1];
} WAPPCDATA, FAR *LPWAPPCDATA;
```

The structure members are as follows:

wVersion
The highest APPC version number supported by the Windows APPC DLL.
szDescription
A descriptive string describing the WinAPPC implementation.

After it makes its last Windows APPC call, an application should call the WinAPPCCleanup routine.

Each Windows APPC implementation must make a **WinAPPCStartup** call before issuing any other Windows APPC calls.

WinAPPCSetBlockingHook

The **WinAPPCSetBlockingHook** function allows a Windows APPC implementation to block APPC function calls by means of a new function. This call is used by Microsoft® Windows® version 3.*x* applications to make blocking calls without blocking the rest of the system. By default in Microsoft® Windows NT®, Windows 95, and Windows 98, blocking calls suspend the calling application's thread until the request is finished. Therefore, if a single-threaded application is targeted at both Windows 2000, Windows NT, Windows 98, Windows 95, and Windows version 3.*x*, and relies on this functionality, it should register a blocking hook even if the default hook will suffice.

```
FARPROC WINAPI WinAPPCSetBlockingHook (
    FARPROC lpBlockFunc
);
```

Parameters

lpBlockFunc
Specifies the procedure instance address of the blocking function to be installed.

Return Values

The return value points to the procedure instance of the previously installed blocking function. The application or library that calls **WinAPPCSetBlockingHook** should save this return value so that it can be restored if needed. (If nesting is not important, the application can simply discard the value returned by **WinAPPCSetBlockingHook** and eventually use WinAPPCUnhookBlockingHook to restore the default mechanism.)

Remarks

A Windows APPC implementation has a default mechanism by which blocking APPC functions are implemented. This function gives the application the ability to execute its own function at blocking time in place of the default function.

The default blocking function is equivalent to:

```
BOOL DefaultBlockingHook (void)  {
    MSG msg;
    /* get the next message if any */
    if ( PeekMessage (&msg,0,0,PM_NOREMOVE)  )  {
        if ( msg.message = = WM_QUIT  )
            return FALSE;   // let app process WM_QUIT
        PeekMessage (&msg,0,0,PM_REMOVE) ;
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    /* TRUE if no WM_QUIT received */
    return TRUE;
}
```

A blocking function must return FALSE if it receives a WM_QUIT message so Windows APPC can return control to the application to process the message and terminate gracefully. Otherwise, the function should return TRUE.

This function is implemented on a per-thread basis. It provides for a particular thread to replace the blocking mechanism without affecting other threads.

The **WinAPPCSetBlockingHook** function is provided to support those applications that require more complex message processing—for example, those employing the multiple document interface (MDI) model.

See Also

WinAPPCIsBlocking, WinAPPCCancelBlockingCall

WinAPPCUnhookBlockingHook

The **WinAPPCUnhookBlockingHook** function removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

```
BOOL WINAPI WinAPPCUnhookBlockingHook(void);
```

Return Values

The return value specifies the outcome of the function. It is nonzero if the default mechanism is successfully reinstalled. The value is zero if the mechanism did not reinstall.

See Also

WinAPPCSetBlockingHook

Host Integration Server 2000 Enhancements to the Windows Environment

This section describes the Microsoft® Host Integration Server 2000-specific extensions to Windows APPC and the Common Service Verb (CSV) API.

The GetAppcConfig function takes a local LU and returns the remote LUs that are accessible to the user through that LU. If left blank, and a default local LU has been configured, the user's default local LU will be used. In all instances, if one of the returned remote LUs is the user's default, this is indicated as such. For more information on configuring default local and remote LUs, see the *Microsoft Host Integration Server 2000 online books*. The call is asynchronous and completion is normally signaled by the posting of a Windows message. However, an alternative completion mechanism is provided for console applications.

The GetAppcReturnCode and GetCsvReturnCode functions convert the primary and secondary return codes in the verb control block (VCB) to a printable string. These functions provide a standard set of error strings for use by applications.

For each extension, this section provides a definition of the function, syntax, returns, and remarks for using the function.

GetAppcConfig

The **GetAppcConfig** function provides an asynchronous entry point for retrieving the remote systems to which a particular local LU can connect.

```
HANDLE WINAPI GetAppcConfig(
    HANDLE hWnd,
    LPSTR pLocalLu,
    LPSTR pMode,
    LPINT pNumRemLu,
    INT iMaxRemLu,
    PSTR pRemLu,
    LPINT pAsyncRetCode
);
```

Parameters

hWnd
Supplied parameter. Contains the handle of the window that is to receive an asynchronous completion message when the call has completed. If non-null, the completion message will be posted to this window handle. In this case, *pAsyncRetCode* (the last parameter) must be null. Asynchronous message completion is the recommended approach for a Windows applications to use this function.

pLocalLu
Supplied parameter. Specifies the address of a buffer containing the local LU name for which information is returned. This local LU name must be specified as follows:

- Nonpadded
- Null-terminated
- ASCII string
- Maximum length of eight bytes (excluding the terminator)

To request that the user's default local LU be used, the buffer should contain eight spaces followed by a null.

pMode
Supplied parameter. Specifies the address of a buffer containing the mode name for which information is returned. In Microsoft® SNA Server version 3.0 and later this parameter is not used, but for compatibility with earlier versions of SNA Server a mode name must be specified as follows:

- Nonpadded
- Null-terminated
- ASCII string
- Maximum length of eight bytes (excluding the terminator)

pNumRemLu
Supplied parameter. Specifies the address of an integer variable that when the function completes will contain the number of remote LUs that would have been returned, had the buffer specified by *pRemLu* been large enough to accommodate all of the remote LUs.

iMaxRemLu
Supplied parameter. Specifies the number of remote LU names that can be held by the buffer indicated by *pRemLu*.

pRemLu
Supplied parameter. Specifies the address of the buffer that will hold the remote LU names after the function completes. The information will be returned as an array of strings. Each remote LU name will be stored in the buffer as follows:

- Nonpadded
- Null-terminated
- ASCII string
- Maximum length of eight bytes (excluding the terminator)

The strings start every nine bytes in the buffer, and thus (*pRemLu* + (*i*–1)*9) gives the start of the *i*th string. In the case where the buffer is too small to hold all the names, only *iMaxRemLu* strings will be returned.

pAsyncRetCode

Supplied parameter. Specifies the address of an integer variable used to store the return code from this function, if the supplied address is non-null. The return codes will be the same as those returned by an asynchronous completion message. While the call is completing, the value of the this variable will be APPC_CFG_PENDING. When this asynchronous call is completed, the value of this variable will contain some return code other than APPC_CFG_PENDING.

This variable is used by polling for completion when asynchronous message completion to a window handle is not used.

Note that if *pAsyncRetCode* is used, *hWnd* must be null.

Return Values

The meaning of the immediate return value depends on whether or not the asynchronous request was accepted. To test for acceptance, evaluate the expression:

(<*Returned Handle*> & APPC_CFG_SUCCESS)

If the expression is FALSE, the request was rejected. The return value is then one of the synchronous return codes in the following list. If the expression is TRUE, the request was accepted, and one of the following cases will apply.

- If *hWnd* was non-null, a completion message will arrive in the following form:

Message parameter	**Description**
hWnd	The handle of the target window. This value is the same as the value passed in hWnd on the initial call.
uMsg	Matches the number returned by a call to **RegisterWindowMessage**, with WinAppcCfg used as the identifying string. This string is available by the **#define WIN_APPC_CFG_COMPLETION_MSG**.
wParam	Matches the HANDLE returned from the initial call. It is used as a correlator.
lParam	Contains one of the asynchronous return codes in the following list.

- If *pAsyncRetCode* was non-null, then the specified integer variable will be set to APPC_CFG_PENDING. After this function completes asynchronously, its value will change to one of the asynchronous return codes listed below.

Synchronous Return Codes

APPC_CFG_ERROR_NO_APPC_INIT

The Windows APPC library needs to be initialized by a call to WinAPPCStartup before calling **GetAPPCConfig** and this has not been done.

APPC_CFG_ERROR_INVALID_HWND

The handle passed in *hWnd* was non-null, yet not a valid window handle.

APPC_CFG_ERROR_BAD_POINTER

The *hWnd* parameter was null, indicating that completion was signaled by setting the integer variable pointed to by *pAsyncRetCode*, but *pAsyncRetCode* was not a valid pointer.

APPC_CFG_ERROR_UNCLEAR_COMPLETION_MODE

Both *hWnd* and *pAsyncCompletion* were non-null, so **GetAPPCConfig** was unable to decide how completion should be signaled.

APPC_CFG_ERROR_TOO_MANY_REQUESTS

Too many **GetAPPCConfig** calls are already being processed (currently, this indicates 16 requests are outstanding). Try the call again after a delay. For the Microsoft® Windows® version 3.*x* system, you must yield during this period.

APPC_CFG_ERROR_GENERAL_FAILURE

An unexpected error occurred, probably of a system nature.

Asynchronous Return Codes

APPC_CFG_SUCCESS_NO_DEFAULT_REMOTE

The configuration information has been retrieved, and either no default remote LU was defined or it was not accessible by the specified local LU.

APPC_CFG_SUCCESS_DEFAULT_REMOTE

The configuration information has been retrieved, and there is a default remote LU that is accessible by the specified local LU.

APPC_CFG_ERROR_NO_DEFAULT_LOCAL_LU

An attempt was made to retrieve remote LUs partnered with the default local LU, but no default local LU was configured.

APPC_CFG_ERROR_BAD_LOCAL_LU

The local LU specified is either not configured, or is not valid for the calling verb.

APPC_CFG_ERROR_GENERAL_FAILURE

An unexpected error occurred, probably of a system nature.

Remarks

WinAPPCStartup must be called before using **GetAPPCConfig**.

Whether an error code represents success or failure can be determined by evaluating either (*RetCode&* APPC_CFG_SUCCESS) to test for success or (*RetCode&* APPC_CFG_FAILURE) to test for failure.

The following code fragment shows how a console application can test completion:

```
while (*pAsyncRetCode == APPC_CFG_PENDING)
{
    sleep(250);
}
```

GetAppcReturnCode

The **GetAppcReturnCode** function converts the primary and secondary return codes in the verb control block to a printable string. This function provides a standard set of error strings for use by APPC applications such as 5250 emulators.

```
int WINAPI GetAppcReturnCode(
    struct appc_hdr FAR * vpb,
    UINT buffer_length,
    unsigned char FAR * buffer_addr
);
```

Parameters

vpb
Supplied parameter. Specifies the address of the verb control block.

buffer_length
Supplied parameter. Specifies the length of the buffer pointed to by *buffer_addr*. The recommended length is 256.

buffer_addr
Supplied parameter. Specifies the address of the buffer that will hold the formatted, null-terminated string.

Return Values

The **GetAppcReturnCode** function returns a positive value on success that indicates the length of the error string passed back in *buffer_addr*.

A return value of zero indicates an error. On Microsoft® Windows 2000, Windows NT®, Windows® 98, and Windows® 95, a call to **GetLastError** provides the actual error return code as follows:

0x20000001
The parameters are invalid; the function could not read from the specified verb control block or could not write to the specified buffer.

0x20000002
The specified buffer is too small.

0x20000003
The APPC string library APPCSTR.DLL (for Windows) or APPCST32.DLL (for Windows 2000, Windows NT, Windows 98, and Windows 95) could not be loaded.

Remarks

The descriptive error string returned in *buffer_addr* does not terminate with a new line character (**\n**).

The descriptive error strings are contained in APPCSTR.DLL (for Windows version 3.*x*) or APPCST32.DLL (for Windows 2000, Windows NT, Windows 98, and Windows 95) and can be customized for different languages.

GetCsvReturnCode

The **GetCsvReturnCode** function converts the primary and secondary return codes in the verb control block to a printable string. This function provides a standard set of error strings for use by applications using CSVs.

```
int WINAPI GetCsvReturnCode(
    struct csv_hdr FAR * vpb,
    UINT buffer_length,
    unsigned char FAR * buffer_addr
);
```

Parameters

vpb
Supplied parameter. Specifies the address of the verb control block.

buffer_length
Supplied parameter. Specifies the length of the buffer pointed to by *buffer_addr*. The recommended length is 256.

buffer_addr
Supplied parameter. Specifies the address of the buffer that will hold the formatted, null-terminated string when the function completes.

Return Values

The **GetCsvReturnCode** function returns a positive value on success that indicates the length of the error string passed back in *buffer_addr*.

A return value of zero indicates an error. On Microsoft® Windows 2000, Windows NT®, Windows 98, and Windows 95, a call to **GetLastError** provides the actual error return code as follows:

0x20000001
The parameters are invalid; the function could not read from the specified verb parameter block or could not write to the specified buffer.

0x20000002
The specified buffer is too small.

0x20000003
The CSV string library CSVSTR.DLL (for Microsoft® Windows® version 3.*x*) or CSVST32.DLL (for Windows NT, Windows 95, and Windows 98) could not be loaded.

Remarks

The descriptive error string returned in *buffer_addr* does not terminate with a newline character (**\n**).

The descriptive error strings are contained in CSVSTR.DLL (for Windows version 3.*x*) or CSVST32.DLL (for Windows NT, Windows 95, and Windows 98) and can be customized for different languages.

Common Service Verbs

This section describes each of the CSVs and provides:

- A definition of the verb.
- The structure that defines the VCB used by the verb. The structure is declared in the WINCSV.H file.
- The parameters (VCB fields) supplied to and returned by the verb. A description of each parameter is provided, along with its possible values and other information.
- Additional information describing the use of the verb.

Most parameters supplied to and returned by CSVs are hexadecimal values. To simplify coding, these values are represented by meaningful symbolic constants, which are established by **#define** statements in the header file WINCSV.H. For example, the **opcode** (operation code) parameter for CONVERT is the hexadecimal value represented by the symbolic constant SV_CONVERT. Use only the symbolic constants when programming CSVs.

CONVERT

The **CONVERT** verb translates an ASCII character string to EBCDIC or an EBCDIC character string to ASCII. The string to be converted is called the source string. The converted string is called the target string.

The following structure describes the verb control block used by the **CONVERT** verb.

```
struct convert {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        direction;
    unsigned char        char_set;
    unsigned short       len;
    unsigned char FAR * source;
    unsigned char FAR * target;
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_CONVERT.

opext
A reserved field.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

direction
Supplied parameter. Specifies the direction of the conversion. To convert from ASCII to EBCDIC, use SV_ASCII_TO_EBCDIC. To convert from EBCDIC to ASCII, use SV_EBCDIC_TO_ASCII.

char_set
Supplied parameter. Specifies the character set to use in converting the source string. Allowed values include SV_A (type A character set), SV_AE (type AE character set), and SV_G (user-defined type G character set).

len
Supplied parameter. Specifies the number of characters to be converted.

This length plus the offset from the beginning of the source or target buffer must not exceed the segment boundary.

source
Supplied parameter. Specifies the address of the buffer containing the character string to be converted.

target
Supplied parameter. Specifies the address of the buffer to contain the converted character string.

This buffer can overlap or coincide with the buffer pointed to by the **source** parameter. In this case, the converted data string overwrites the source data string.

Return Codes

SV_OK
Primary return code; the verb executed successfully.

SV_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

SV_CONVERSION_ERROR

Secondary return code; one or more characters in the source string were not found in the conversion table. These characters were converted to nulls (0x00). The verb still executed.

SV_INVALID_CHARACTER_SET

Secondary return code; the **char_set** parameter contained an invalid value.

SV_INVALID_DATA_SEGMENT

Secondary return code; the data buffer containing the source or target string did not fit in one segment, or the target segment was not a read/write segment. This applies only to the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, and OS/2 operating systems.

SV_INVALID_DIRECTION

Secondary return code; the direction contained an invalid value.

SV_INVALID_FIRST_CHARACTER

Secondary return code; the first character of a type A source string was invalid.

SV_TABLE_ERROR

Secondary return code; one of the following occurred:

- The file containing the user-written type G conversion table was not specified by the environment variable **CSVTBLG**.
- The table was not in the correct format.
- The file specified by the **CSVTBLG** variable was not found.

SV_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

SV_INVALID_VERB
Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.

SV_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

SV_UNEXPECTED_DOS_ERROR
Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT, Microsoft® Windows 98, Microsoft® Windows 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

Remarks

The type A character set consists of:

- Uppercase letters.
- Numerals 0 through 9.
- Special characters $, #, @, and space.

This character set is supported by a system-supplied type A conversion table.

The first character of the source string must be an uppercase letter or the special character $, #, or @. Spaces are allowed only in trailing positions. Lowercase ASCII letters are translated to uppercase EBCDIC letters when the direction is ASCII to EBCDIC.

The type AE character set consists of:

- Uppercase letters.
- Lowercase letters.
- Numerals 0 through 9.
- Special characters $, #, @, period, and space.

This character set is supported by a system-supplied type AE conversion table.

The first character of the source string can be any character in the character set, except the space. Spaces are allowed only in trailing positions.

During conversion, embedded blanks (including blanks in the first position) are converted to 0x00. Although such a conversion will complete, CONVERSION_ERROR is returned as the secondary return code, indicating that the CSV library has completed an irreversible conversion on the supplied data.

For Windows 2000 or Windows NT, a description of **COMTBLG** should point to the Windows 2000 or Windows NT registry under **\SnaBase\Parameters\Client**. For Windows version 3.*x*, the fully qualified file name for a type G conversion table must be the **COMTBLG**=*filename* entry of the [WNAP] section in the WIN.INI file. Entries in this section of the WIN.INI file are used in all places where MS-DOS and OS/2 use environment variables. (If the file is not found, the system returns the parameter check SV_TABLE_ERROR.)

The data for a type G conversion table must be an ASCII file 32 lines long. Each line must consist of 32 hexadecimal digits, representing 16 characters, and be terminated by a carriage return and line feed. The first 16 lines (256 characters) specify the EBCDIC characters to which ASCII characters are converted; the remaining 16 lines specify the ASCII characters to which EBCDIC characters are converted.

The hexadecimal digits A through F can be either uppercase or lowercase. However, you may want to make these digits uppercase to ensure compatibility with IBM ES for OS/2 version 1.0.

> **Note** You can use GET_CP_CONVERT_TABLE to build a type G user-written conversion table in memory, and then store the table in a file.

COPY_TRACE_TO_FILE

The **COPY_TRACE_TO_FILE** verb concatenates individual API/link service trace files to form a single file.

The following structure describes the verb control block used by the **COPY_TRACE_TO_FILE** verb.

```
struct copy_trace_to_file {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        reserv3[8];
    unsigned char        file_name[64];
    unsigned char        file_option;
    unsigned char        reserv4[12];
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_COPY_TRACE_TO_FILE.

opext
A reserved field.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

reserv3
A reserved field.

file_name
Supplied parameter. Specifies the name of the file to which trace data is to be copied. This parameter is a 64-byte character string, and it can include a path. If the name is fewer than 64 bytes, use spaces to pad it on the right.

file_option
Supplied parameter. Specifies the output file copy option:

- Use SV_NEW to copy the trace only if the specified file does not already exist.
- Use SV_OVERWRITE to copy the trace to an existing file, overwriting the current data. The size of the file is increased if necessary; and the file is created if it does not already exist.

reserv4
The address at which supplied data resides.

Return Codes

SV_OK
Primary return code; the verb executed successfully.

SV_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

SV_INVALID_FILE_OPTION

Secondary return code; a value other than SV_NEW or SV_OVERWRITE was specified for **file_option**.

SV_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

SV_COPY_TRACE_IN_PROGRESS

Secondary return code; a previously issued **COPY_TRACE_TO_FILE** verb is still in progress.

SV_TRACE_FILE_EMPTY

Secondary return code; there is no data in the trace files.

SV_TRACE_NOT_STOPPED

Secondary return code; a trace was in progress when the verb was issued.

SV_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

SV_FILE_ALREADY_EXISTS
Primary return code; when the SV_NEW file option was used, the file name specified was the name of an existing file.

SV_INVALID_VERB
Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.

SV_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

SV_OUTPUT_DEVICE_FULL
Primary return code; there is insufficient space on the device where the output file resides. Retry the operation after freeing additional disk space.

SV_UNEXPECTED_DOS_ERROR
Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

Remarks

There are two API/link-service trace files. The files are used alternately; tracing switches from one file to the other when one file is full (larger than 250K). When **COPY_TRACE_TO_FILE** is called, these trace files are concatenated and copied to a single file, the name of which is specified as a parameter to the call.

API/link-service tracing is stopped before issuing the verb, and restarted after the copy is complete. The trace files are reset when this verb is successfully completed.

DEFINE_TRACE

The **DEFINE_TRACE** verb enables or disables tracing for specified APIs and controls the amount of tracing.

The following structure describes the verb control block used by the **DEFINE_TRACE** verb.

```
struct define_trace {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        reserv3[8];
    unsigned char        dt_set;
    unsigned char        appc;
    unsigned char        reserv4;
    unsigned char       srpi;
    unsigned char       sdlc;
    unsigned char       tkn_rng_dlc;
    unsigned char       pcnet_dlc;
    unsigned char       dft;
    unsigned char       acdi;
    unsigned char       reserv5;
    unsigned char       ehllapi;
    unsigned char       x25_api;
    unsigned char       x25_dlc;
    unsigned char       twinax;
    unsigned char       reserv6;
    unsigned char       lua_api;
    unsigned char       etherand;
    unsigned char       subsym;
    unsigned char       reserv7[8];
    unsigned char       reset_trc;
    unsigned short      trunc;
    unsigned short      strg_size;
    unsigned char       reserv8;
    unsigned char       phys_link[8];
    unsigned char       reserv9[56];
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_DEFINE_TRACE.

opext
A reserved field.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

reserv3
A reserved field.

dt_set
Supplied parameter. Sets the trace state.

- Use SV_ON to enable tracing for a particular API if the parameter pertaining to the API (such as **appc** or **comm_serv**) is set to SV_CHANGE.
- Use SV_OFF to disable tracing for a particular API if the parameter pertaining to the API is set to SV_CHANGE.

appc

Supplied parameter. Indicates whether tracing of APPC is desired.

- Use SV_CHANGE to enable or disable tracing for APPC, depending on the **dt_set** parameter.
- Use SV_IGNORE to leave tracing in its current state for APPC.

The allowed values turn bit 0 on or off; bits 1 through 7 are reserved.

reserv4
A reserved field.
srpi
Supplied parameter. Indicates whether tracing of SRPI is desired.

- Use SV_CHANGE to enable or disable tracing for APPC, depending on the **dt_set** parameter.
- Use SV_IGNORE to leave tracing in its current state for APPC.

sdlc
A reserved field.
tkn_rng_dlc
A reserved field.
pcnet_dlc
A reserved field.
dft
A reserved field.
acdi
A reserved field.
reserv5
A reserved field.
comm_serv
Supplied parameter. Indicates whether tracing of COMM_SERV_API is desired.

- Use SV_CHANGE to enable or disable tracing for APPC, depending on the **dt_set** parameter.
- Use SV_IGNORE to leave tracing in its current state for APPC.

ehllapi
A reserved field.
x25_api
A reserved field.
x25_dlc
A reserved field.
twinax
A reserved field.
reserved6
A reserved field.
lua_api
A reserved field.
etherand
A reserved field.
subsym
A reserved field.
reserved7
A reserved field.
reset_trc
Supplied parameter. Indicates whether the trace file pointer should be reset.

- Use SV_NO to not reset the trace file pointer to the start of the trace file. Previous trace records are not overwritten.
- Use SV_YES to reset the trace file pointer to the start of the trace file. Previous trace records are overwritten.

trunc
Supplied parameter. Specifies the maximum number of bytes for each trace record. Excess bytes are truncated. Set this value to zero if you do not want truncation.
strg_size
A reserved field.
reserved8

A reserved field.

phys_link

A reserved field.

reserved9

A reserved field.

Return Codes

SV_OK

Primary return code; the verb executed successfully.

SV_PARAMETER_CHECK

Primary return code; the verb did not execute because of a parameter error.

SV_INVALID_RESET_TRACE

Secondary return code; the **reset_trc** parameter contained an invalid value.

SV_INVALID_SET

Secondary return code; the **dt_set** parameter contained an invalid value.

SV_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

SV_COPY_TRACE_IN_PROGRESS

Secondary return code; a previously issued COPY_TRACE_TO_FILE is still in progress. Traces cannot be active while using **DEFINE_TRACE**.

SV_COMM_SUBSYSTEM_NOT_LOADED

Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

SV_INVALID_VERB

Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.

SV_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

SV_UNEXPECTED_DOS_ERROR

Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

Remarks

For information on how to run and use traces, see the appropriate manual for your product.

GET_CP_CONVERT_TABLE

The **GET_CP_CONVERT_TABLE** verb creates and returns a 256-byte conversion table to translate character strings from a source code page to a target code page.

The following structure describes the verb control block used by the **GET_CP_CONVERT_TABLE** verb.

```
struct get_cp_convert_table {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        reserv3[8];
    unsigned short       source_cp;
    unsigned short       target_cp;
    unsigned char FAR * conv_tbl_addr;
    unsigned char        char_not_fnd;
    unsigned char        substitute_char;
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_GET_CP_CONVERT_TABLE.

opext
A reserved field.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

reserv3
A reserved field.

source_cp
Supplied parameter. Specifies the source code page from which characters are converted. The allowed code pages (decimal values) are as follows:

- ASCII 437, 850, 860, 863, 865
- EBCDIC 037, 273, 277, 278, 280, 284, 285, 297, 500

User-defined code pages in the range from 65280 through 65535 are also allowed.

ASCII code pages are sometimes referred to as PC code pages; EBCDIC code pages are sometimes referred to as host code pages.

target_cp
Supplied parameter. Specifies the target code page to which characters are converted. For allowed code pages, see the preceding definition for **source_cp**.

conv_tbl_addr
Supplied parameter. Specifies the address of the buffer to contain the 256-byte conversion table. The buffer must be in a writable segment and long enough to contain the table.

char_not_fnd
Supplied parameter. Specifies the action to take if a character in the source code page does not exist in the target code page:

- Use SV_ROUND_TRIP to store a unique value in the conversion table for each source code page character.
- Use SV_SUBSTITUTE to store a substitute character (specified by **substitute_char**) in the conversion table.

substitute_char
Supplied parameter. Specifies the character to store in the conversion table when a character from the source code page has no equivalent in the target code page.

Return Codes

SV_OK
Primary return code; the verb executed successfully.
SV_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.
SV_INVALID_CHAR_NOT_FOUND
Secondary return code; the **char_not_fnd** parameter contained an invalid value.
SV_INVALID_DATA_SEGMENT
Secondary return code; the 256-byte area specified for the conversion table extended beyond the segment boundary, or the segment was not writable.
SV_INVALID_SOURCE_CODE_PAGE
Secondary return code; the code page specified by **source_cp** is not supported.
SV_INVALID_TARGET_CODE_PAGE
Secondary return code; the code page specified by **target_cp** is not supported.
SV_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
SV_INVALID_VERB
Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.
SV_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
SV_UNEXPECTED_DOS_ERROR
Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

Remarks

The type A character set consists of:

- Uppercase letters.
- Numerals 0 through 9.
- Special characters $, #, @, and space.

This character set is supported by a system-supplied type A conversion table.

The first character of the source string must be an uppercase letter or the special character $, #, or @. Spaces are allowed only in trailing positions. Lowercase ASCII letters are translated to uppercase EBCDIC letters when the direction is ASCII to EBCDIC.

The type AE character set consists of:

- Uppercase letters.
- Lowercase letters.
- Numerals 0 through 9.
- Special characters $, #, @, period, and space.

This character set is supported by a system-supplied type AE conversion table.

The first character of the source string can be any character in the character set except the space.

During conversion, embedded blanks (including blanks in the first position) are converted to 0x00. Although such a conversion will complete, CONVERSION_ERROR is returned as the secondary return code, indicating that the CSV library has completed an irreversible conversion on the supplied data.

For Windows 2000 or Windows NT, a description of **COMTBLG** should point to the Windows 2000 or Windows NT registry under **\SnaBase\Parameters\Client**. For the OS/2 operating system, the directory and file containing the table must be specified by the environment variable **COMTBLG**. (If the file is not found, the system returns the SV_TABLE_ERROR parameter check.).

The SV_ROUND_TRIP value for **char_not_fnd** is useful only if you build a second conversion table to convert between the same two code pages in the reverse direction. If you specify the SV_ROUND_TRIP value in building both conversion tables, any character translated from one code page to the other and then back will be unchanged.

When using the SV_SUBSTITUTE value for **char_not_fnd**, converting the translated character string back to the original code page will not necessarily re-create the original character string.

Use **substitute_char** only if **char_not_fnd** is set to SV_SUBSTITUTE.

The value stored in the conversion table is the ASCII value associated with the character. If the table is used for conversion from ASCII to EBCDIC, the character that appears in the converted string is the character associated with the numeric EBCDIC value rather than ASCII.

For example, if you supply the underscore (_) character (ASCII value F6) while creating an ASCII to EBCDIC conversion table, the character that appears in the converted strings will be 6, the character associated with the value F6 in EBCDIC. To use the _ character as the substitute character in an ASCII to EBCDIC conversion table, you should supply the value E1 (the value associated with the _ character in EBCDIC) rather than the actual character.

A code page is a table that associates specific ASCII or EBCDIC values with specific characters. If a character from the source code page does not exist in the target code page, the translated (target) string differs from the original (source) string.

LOG_MESSAGE

For OS/2 only, the **LOG_MESSAGE** verb records a message in the error log file and optionally displays the message on the user's screen. This verb is included for compatibility with existing applications.

The following structure describes the verb control block used by the **LOG_MESSAGE** verb.

```
struct log_message {
    unsigned short       opcode;
    unsigned char        opext;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned short       msg_num;
    unsigned char        orignтr_id[8];
    unsigned char        msg_file_name[3];
    unsigned char        msg_act;
    unsigned short       msg_ins_len;
    unsigned char FAR * msg_ins_ptr;
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_LOG_MESSAGE.

opext
A reserved field.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

msg_num
Supplied parameter. Specifies the number of the message in the message file specified by **msg_file_name**.

orignтr_id
Supplied parameter. Specifies the name of the component issuing **LOG_MESSAGE** or an 8-byte, user-supplied string.

msg_file_name
Supplied parameter. Specifies the name of the file containing the message to be logged.

msg_act
Supplied parameter. Specifies the action to be taken when processing the message:

- Use SV_INTRV to log the intervention with a severity level of 12 and display the message on the user's screen. The user must press a key to remove the message from the screen.
- Use SV_NO_INTRV to log the intervention with a severity level of 12 but not display the message.

msg_ins_len
Supplied parameter. Specifies the length of data to be inserted into the message. Set this parameter to zero if no data is to be inserted.

msg_ins_ptr
Supplied parameter. Specifies the address of the data to be inserted into the message.

Use this parameter only if **msg_ins_len** is greater than zero.

Return Codes

SV_OK
Primary return code; the verb executed successfully.

SV_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

SV_INVALID_DATA_SEGMENT
Secondary return code; the data that was to be inserted into the message extended beyond the segment boundary.

SV_INVALID_MESSAGE_ACTION
Secondary return code; the **msg_act** parameter contained an invalid value.
SV_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
SV_INVALID_VERB
Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.
SV_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
SV_UNEXPECTED_DOS_ERROR
Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

Remarks

The value for **msg_file_name** must be three characters long. Pad with spaces if necessary. The .MSG extension is added automatically.

The total length of **msg_ins_len**, including header information (40 bytes), message text, and inserted data, should not exceed 256 bytes. If the length is greater than 256 bytes, the communication system will attempt to log only the header information and inserted text; the message text will be left out.

When you create the log message file, you can specify where in the message the additional data is to be inserted. Further information is provided below.

The data for **msg_ins_ptr** consists of a series of up to nine null-terminated strings. (Because IBM OS/2 ES version 1.0 supports only three data strings, you may want to limit the inserted text to three strings to ensure compatibility.)

Creating a Message File

If you want to create your own message file, you must use the utility MKMSGF. This utility can also run with MS-DOS.

The first three characters of the message number must match the three-character name of the log message file. These three characters are declared at the top of the file as well.

The system finds the message file as follows:

- If you use your own message file, the system assumes the file is in the same directory as your program's executable file.
- If you use the default message file, COM.MSG, the system finds the file automatically, provided the SnaBase for Microsoft® Host Integration Server 2000 or SNA Server is loaded.
- If you use the default message file without loading the previously-mentioned software, the system expects DPATH to indicate the path to the message file. This applies only to the Windows version 3.*x* and OS/2 operating systems.

TRANSFER_MS_DATA

The **TRANSFER_MS_DATA** verb builds an SNA request unit containing Network Management Vector Transport (NMVT) data. The verb can send the NMVT data to NetView for centralized problem diagnosis and resolution. The data is logged in the local audit file.

The following structure describes the verb control block used by the **TRANSFER_MS_DATA** verb.

```
struct transfer_ms_data {
    unsigned short       opcode;
    unsigned char        data_type;
    unsigned char        reserv2;
    unsigned short       primary_rc;
    unsigned long        secondary_rc;
    unsigned char        options;
    unsigned char        reserv3;
    unsigned char        orignTr_id[8];
    unsigned short       dlen;
    unsigned char FAR * dptr;
};
```

Members

opcode
Supplied parameter. The verb identifying the operation code, SV_TRANSFER_MS_DATA.

data_type
Supplied parameter. Specifies the type of data provided by this verb:

- Use SV_NMVT to generate an NMVT (including the NS header, the major network management vector, and subvectors).
- Use SV_ALERT_SUBVECTORS to generate an RU containing data for an alert in the appropriate format, without the NS header or major NMVT vector.
- Use SV_PDSTATS_SUBVECTORS to generate an RU containing data for problem determination statistics in the appropriate format, without the NS header or major NMVT vector.
- Use SV_USER_DEFINED to generate user-defined data; this data is recorded in the error log but cannot be sent on the systems services control point-physical unit (SSCP-PU) session on the connection configured for diagnostics.

reserv2
A reserved field.

primary_rc
Returned parameter. Specifies the primary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

secondary_rc
Returned parameter. Specifies the secondary return code set by APPC at the completion of the verb. The valid return codes vary depending on the APPC verb issued. See Return Codes for valid error codes for this verb.

options
Supplied parameter. Specifies the desired options by turning individual bits on or off. (Bits 1, 2, and 3 are ignored if **data_type** is set to SV_USER_DEFINED.) See the Remarks section.

orignTr_id
Supplied parameter. Specifies the name of the component issuing **TRANSFER_MS_DATA**. This parameter is optional. Set it to 0x00 if you want the system to ignore it.

dlen
Supplied parameter. Specifies the length of data to be supplied to this verb. The total length of the data (user-supplied data and any added headers or subvectors) must fit into one RU. The maximum RU length is 512 bytes.

dptr
Supplied parameter. Specifies the address of the data to be sent.

Return Codes

SV_OK
Primary return code; the verb executed successfully.

SV_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

SV_DATA_EXCEEDS_RU_SIZE

Secondary return code; the data to be sent was too long. The length of the user-supplied data plus headers and added subvectors must fit in a single RU that is not more than 512 bytes long.

SV_INVALID_DATA_SEGMENT

Secondary return code; the buffer pointed to by **dptr** was not a readable segment or extended beyond the segment boundary.

SV_INVALID_DATA_TYPE

Secondary return code; the **data_type** parameter contained an invalid value.

SV_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

SV_SSCP_PU_SESSION_NOT_ACTIVE

Secondary return code; the NMVT was not sent; either the SSCP-PU session was not active, the node configured to receive diagnostic information was not active, or no network management connection was configured.

SV_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
SV_INVALID_VERB
Primary return code; the **opcode** parameter did not match the operation code of any verb. No verb executed.
SV_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
SV_UNEXPECTED_DOS_ERROR
Primary return code; one of the following conditions occurred:

- The Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, Microsoft® Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows **SendMessage** function call, rather than the more common Windows **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

SV_CANCELLED
Primary return code; this code is returned for an asynchronous verb when it has been shut down by a WinCSVCleanup call.
SV_SERVER_RESOURCE_NOT_FOUND
Primary return code; no communication server was found that could provide the requested function.
SV_SERVER_RESOURCES_LOST
Primary return code; the communications server that was providing the function was lost due to a connection failure.

SV_SERVER_CONN_FAILURE

Secondary return code; the connection to the server was lost due to physical path problems; for example, the server may have been powered off.

SV_THREAD_BLOCKING
Primary return code; this verb exceeds the maximum number of simultaneous synchronous verbs allowed.

Remarks

To specify options, turn bits on or off as follows:

Bit	Description
0	TIME_STAMP_SUBVECTOR. Adds date/time subvector to data. Allowed values include SV_ADD and SV_NO_ADD.
1	PRODUCT_SET_ID_SUBVECTOR. Adds Product_Set_ID subvector to data. This allows network management services to identify the sender of an alert. Allowed values include SV_ADD and SV_NO_ADD.
2	SSCP_PU_SESSION. Sends the data on the SSCP-PU session on the connection configured for diagnostics if the session is active. (The data is added to the error log regardless of whether it is sent on the session or whether SV_STATE_CHECK or SV_COMM_SUBSYSTEM_NOT_LOADED is returned.) Allowed values include SV_SEND and SV_NO_SEND.

3	LOCAL_LOGGING. Logs local alerts that are retrieved from the error log and forwarded to the host. This option is valid only when **data_type** SV_NMVT or **data_type** SV_ALERT_SUBVECTORS with option SV_SEND is specified. Allowed values include SV_LOG and SV_NO_LOG.
4 through 7	Reserved

CSV Extensions for the Windows Environment

This section describes API extensions to the Windows Common Service Verb (CSV) API. The extensions described in this section have been designed for all implementations and versions of the Microsoft® Windows® graphical environment, version 3.0 and later. They provide support for maximum Windows programming compatibility and optimum application performance in both 16-bit and 32-bit operating environments.

Windows CSV allows multithreaded Windows-based processes. Multithreading is the running of several processes in rapid sequence within a single program. A process contains one or more threads of execution. The 16-bit Windows environment is not multithreaded. In this instance, a task corresponds to a process with a single thread. All references to threads in this document refer to actual threads in multithreaded Windows environments.

For each extension, this section provides a definition of the function, syntax, returns, and remarks for using the function.

WinAsyncCSV

The **WinAsyncCSV** function provides an asynchronous entry point for TRANSFER_MS_DATA only. If this function is used for any other verb, the behavior will be synchronous. Use this function instead of the blocking version of the verb if you run your application under Microsoft® Windows® version 3.*x*.

```
HANDLE WINAPI WinAsyncCSV(
    HWND hWnd,
    long lpVcb
);
```

Parameters

hWnd
Handle of window to receive message.
lpVcb
Pointer to the verb control block.

Return Values

The return value specifies whether the asynchronous resolution request was successful. If the function was successful, the return value is an asynchronous task handle. If the function was not successful, a zero is returned.

Remarks

When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinAsyncCSV" as the input string. The *wParam* argument contains the asynchronous task handle returned by the original function call. The *lParam* argument contains the original VCB pointer and can be dereferenced to determine the final return code.

If the function returns successfully, a "WinAsyncCSV" message will be posted to the application when the operation completes or the conversation is canceled.

WinCSVCleanup

The **WinCSVCleanup** function terminates and deregisters an application from a Windows CSV implementation.

```
BOOL WINAPI WinCSVCleanup(void);
```

Return Values

The return value specifies whether the deregistration was successful. If the value is nonzero, the application was successfully deregistered. The application was not deregistered if a value of zero is returned.

Remarks

Use **WinCSVCleanup** to indicate deregistration of a Windows CSV application from a Windows CSV implementation. This function can be used, for example, to free up resources allocated to the specific application.

WinCSVStartup

The **WinCSVStartup** function allows an application to specify the version of Windows CSV required and to retrieve details of the specific Windows CSV implementation. This function must be called by an application to register itself with a Windows CSV implementation before issuing any further Windows CSV calls.

```
int WINAPI WinCSVStartup(
    WORD wVersionRequired,
    LPWCSVDATA lpwcsvdata
);
```

Parameters

wVersionRequired
Specifies the version of Windows CSV support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number. The current version of the Windows CSV API is 1.0.

lpwcsvdata
A pointer to the CSV data structure. The **CSVDATA** structure is defined as follows:

```
typedef struct tagWCSVDATA {
....WORD wVersion;
    char szDescription[WCSVDESCRIPTION_LEN+1];
}  CSVDATA, FAR * LPWCSVCDATA;
```

where **WCSVDESCRIPTION** is defined to be 127 and the structure members are as follows:

wVersion

The version of Windows CSV supported. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

szDescription

A description string identifying the vendor of the Windows CSV DLL.

This **CVSDATA** structure provides information about the underlying Windows CSV DLL implementation. The first *wVersion* field has the same structure as the *wVersionRequired* parameter, and the *szDescription* field contains a string identifying the vendor of the Windows CSV DLL. The description field is only meant to provide a display string for the application and should not be used to programmatically distinguish between Windows CSV implementations.

Return Values

The return value specifies whether the application was registered successfully and whether the Windows CSV implementation can support the specified version number. If the value is zero, it was registered successfully. Otherwise, the return value is one of the following:

WCSVSYSNOTREADY
Indicates that the underlying network subsystem is not ready for network communication.

WCSVVERNOTSUPPORTED
The version of Windows CSV support requested is not provided by this particular Windows CSV implementation.

WCSVINVALID
The Windows CSV version specified by the application is not supported by this DLL.

Remarks

To support future Windows CSV implementations and applications that may have functionality differences from Windows CSV version 1.0, a negotiation takes place in **WinCSVStartup**. An application passes to **WinCSVStartup** the Windows CSV version that it can use. If this version is lower than the lowest version supported by the Windows CSV DLL, the DLL cannot support the application and **WinCSVStartup** fails. If the version is not lower, however, the call succeeds and returns the highest version of Windows CSV supported by the DLL. If this version is lower than the lowest version supported by the application, the application either fails its initialization or attempts to find another Windows CSV DLL on the system.

This negotiation allows both a Windows CSV DLL and a Windows CSV application to support a range of Windows CSV versions. An application can successfully use a DLL if there is any overlap in the versions. The following table illustrates how **WinCSVStartup** works in conjunction with different application and DLL versions.

Application versions	DLL versions	To WinCSVStartup	From WinCSVStartup	Result
1.0	1.0	1.0	1.0	Use 1.0
1.0, 2.0	1.0	2.0	1.0	Use 1.0
1.0	1.0, 2.0	1.0	2.0	Use 1.0
1.0	2.0, 3.0	1.0	WCSVINVALID	Fail
2.0, 3.0	1.0	3.0	1.0	App Fails
1.0, 2.0, 3.0	1.0, 2.0, 3.0	3.0	3.0	Use 3.0

After making its last Windows CSV call, an application should call WinCSVCleanup.

Each Windows CSV implementation must make a **WinCSVStartup** call before issuing any other Windows CSV calls. Consequently, this function can be used for initialization purposes.

Common APPC Return Codes

This section describes the primary and, if applicable, secondary return codes for the APPC verbs. The return codes are listed in hexadecimal order.

Primary APPC Return Codes

0000

AP_OK
The verb executed successfully.

0001

AP_PARAMETER_CHECK
The verb did not execute because of a parameter error.

0002

AP_STATE_CHECK
The verb did not execute because it was issued in an invalid state.

0003

AP_ALLOCATION_ERROR
APPC failed to allocate a conversation. The conversation state is set to RESET.

This code can be returned through a verb issued after ALLOCATE or MC_ALLOCATE.

0005

AP_DEALLOC_ABEND (for a mapped conversation)
The conversation has been deallocated for one of the following reasons:

- The partner TP issued MC_DEALLOCATE with **dealloc_type** set to AP_ABEND.
- The partner TP encountered an ABEND, causing the partner LU to send an **MC_DEALLOCATE** request.

0006

AP_DEALLOC_ABEND_PROG (for a basic conversation)
The conversation has been deallocated for one of the following reasons:

- The partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG.
- The partner TP encountered an ABEND, causing the partner LU to send a **DEALLOCATE** request.

0007

AP_DEALLOC_ABEND_SVC (for a basic conversation)
The conversation has been deallocated because the partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_SVC.

0008

AP_DEALLOC_ABEND_TIMER (for a basic conversation)
The conversation has been deallocated because the partner TP issued DEALLOCATE with **dealloc_type** set to AP_ABEND_TIMER.

0009

AP_DEALLOC_NORMAL
The partner TP has deallocated the conversation without requesting confirmation.

000C

AP_PROG_ERROR_NO_TRUNC
The partner TP has issued one of the following verbs while the conversation was in SEND state:

- SEND_ERROR with **err_type** set to AP_PROG
- MC_SEND_ERROR_sna_MC_SEND_ERROR_appc

Data was not truncated.

000F

AP_CONV_FAILURE_RETRY
The conversation was terminated because of a temporary error. Restart the TP to see if the problem occurs again. If it does, the

system administrator should examine the error log to determine the cause of the error.

0010

AP_CONV_FAILURE_NO_RETRY
The conversation was terminated because of a permanent condition, such as a session protocol error. The system administrator should examine the system error log to determine the cause of the error. Do not retry the conversation until the error has been corrected.

0011

AP_SVC_ERROR_NO_TRUNC
While in SEND state, the partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC. Data was not truncated.

0012

AP_PROG_ERROR_TRUNC/AP_SVC_ERROR_TRUNC
In SEND state, after sending an incomplete logical record, the partner TP issued SEND_ERROR. The local TP may have received the first part of the logical record.

0013

AP_SVC_ERROR_PURGING
The partner TP (or partner LU) issued SEND_ERROR with **err_type** set to AP_SVC while in RECEIVE, PENDING_POST (Microsoft® Windows NT®, Microsoft® Windows® 95, and OS/2 only), CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state. Data sent to the partner TP may have been purged.

0014

AP_UNSUCCESSFUL
No data is immediately available from the partner TP.

0017

AP_CNOS_LOCAL_RACE_REJECT
APPC is currently processing a CNOS verb issued by a local LU.

0018

AP_CNOS_PARTNER_LU_REJECT
The partner LU rejected a CNOS request from the local LU.

0019

AP_CONVERSATION_TYPE_MIXED
The TP has issued both basic and mapped conversation verbs. Only one type can be issued in a single conversation.

0021

AP_CANCELED
The local TP issued one of the following verbs, which canceled RECEIVE_AND_POST or MC_RECEIVE_AND_POST:

- DEALLOCATE with **dealloc_type** set to AP_ABEND_PROG, AP_ABEND_SVC, or AP_ABEND_TIMER
- MC_DEALLOCATE with **dealloc_type** set to AP_ABEND
- SEND_ERROR or MC_SEND_ERROR
- TP_ENDED_sna_TP_ENDED_appc

Issuing one of these verbs causes the semaphore to be cleared.

F002

AP_TP_BUSY
The local TP has issued a call to APPC while APPC was processing another call for the same TP. This can occur if the local TP has multiple threads, and more than one thread is issuing APPC calls using the same **tp_id**.

F003

AP_COMM_SUBSYSTEM_ABENDED
Indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.

- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

The system administrator should examine the error log to determine the reason for the ABEND.

F004

AP_COMM_SUBSYSTEM_NOT_LOADED
A required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

F005

AP_CONV_BUSY
There can only be one outstanding conversation verb at a time on any conversation.

F006

AP_THREAD_BLOCKING
The calling thread is already in a blocking call.

F008

AP_INVALID_VERB_SEGMENT
The VCB extended beyond the end of the data segment.

F011

AP_UNEXPECTED_DOS_ERROR
The operating system returned an error to APPC while processing an APPC call from the local TP. The operating system return code is returned through the **secondary_rc**. It appears in Intel byte-swapped order. If the problem persists, consult the system administrator.

F015

AP_STACK_TOO_SMALL
The stack size of the application is too small to execute the verb. Increase the stack size of your application.

F020

AP_INVALID_KEY
The supplied **key** was incorrect.

Secondary APPC Return Codes

00000000

AP_CNOS_ACCEPTED
APPC accepts the session lines and responsibility as specified.

00000001

AP_BAD_TP_ID
The value of **tp_id** did not match a TP identifier assigned by APPC.

00000002

AP_BAD_CONV_ID
The value of **conv_id** did not match a conversation identifier assigned by APPC.

00000003

AP_BAD_LU_ALIAS
APPC cannot find the specified **lu_alias** among those defined.

000000C4

AP_RCV_IMMD_BAD_FILL (for a basic conversation)
The **fill** parameter was set to an invalid value.

00000004

AP_ALLOCATION_FAILURE_NO_RETRY
The conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

00000005

AP_ALLOCATION_FAILURE_RETRY
The conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

00000006

AP_INVALID_DATA_SEGMENT
The PIP data was longer than the allocated data segment, or the address of the PIP data buffer was wrong.

00000007

AP_CNOS_NEGOTIATED
APPC accepts the session limits and responsibility as negotiable by the partner LU. Values that can be negotiated are:

plu_mode_session_limit

min_conwinners_source

min_conwinners_target

responsible

drain_target

000000D7

AP_BAD_RETURN_STATUS_WITH_DATA
The specified **rtn_status** value was not recognized by APPC.

00000011

AP_BAD_CONV_TYPE (for a basic conversation)
The value specified for **conv_type** was invalid.

00000012

AP_BAD_SYNC_LEVEL

The value specified for **sync_level** was invalid.

00000013

AP_BAD_SECURITY
The value specified for **security** was invalid.

00000014

AP_BAD_RETURN_CONTROL
The value specified for **rtn_ctl** was invalid.

00000016

AP_PIP_LEN_INCORRECT
The value of **pip_dlen** was greater than 32767.

00000017

AP_NO_USE_OF_SNASVCMG (for a mapped conversation)
SNASVCMG is not a valid value for **mode_name**.

00000018

AP_UNKNOWN_PARTNER_MODE
The value specified for **mode_name** was invalid.

00000031

AP_CONFIRM_ON_SYNC_LEVEL_NONE
The local TP attempted to use CONFIRM or MC_CONFIRM in a conversation with a synchronization level of AP_NONE. The synchronization level, established by ALLOCATE or MC_ALLOCATE, must be AP_CONFIRM_SYNC_LEVEL.

00000032

AP_CONFIRM_BAD_STATE
The conversation was not in SEND state.

00000033

AP_CONFIRM_NOT_LL_BDY
The conversation for the local TP was in SEND state, and the local TP did not finish sending a logical record.

00000051

AP_DEALLOC_BAD_TYPE
The **dealloc_type** parameter was not set to a valid value.

00000052

AP_DEALLOC_FLUSH_BAD_STATE

The conversation was not in SEND state and the TP attempted to flush the send buffer. This attempt occurred because the value of **dealloc_type** was AP_FLUSH or because the value of **dealloc_type** was AP_SYNC_LEVEL and the synchronization level of the conversation was AP_NONE. In either case, the conversation must be in SEND state.

00000053

AP_DEALLOC_CONFIRM_BAD_STATE
The conversation was not in SEND state, and the TP attempted to flush the send buffer and send a confirmation request.

00000055

AP_DEALLOC_NOT_LL_BDY (for a basic conversation)
The conversation was in SEND state, and the TP did not finish sending a logical record. The **dealloc_type** parameter was set to AP_SYNC_LEVEL or AP_FLUSH.

00000057

AP_DEALLOC_LOG_LL_WRONG
The LL field of the GDS error log variable did not match the actual length of the log data.

00000061

AP_FLUSH_NOT_SEND_STATE
The conversation was not in SEND state.

000000A1

AP_P_TO_R_INVALID_TYPE
The **ptr_type** parameter was not set to a valid value.

000000A2

AP_P_TO_R_NOT_LL_BDY
The local TP did not finish sending a logical record.

000000A3

AP_P_TO_R_NOT_SEND_STATE
The conversation was not in SEND state.

000000B1

AP_RCV_AND_WAIT_BAD_STATE
The conversation was not in RECEIVE or SEND state when the TP issued this verb.

000000B2

AP_RCV_AND_WAIT_NOT_LL_BDY (for a basic conversation)
The conversation was in SEND state; the TP began but did not finish sending a logical record.

000000B5

AP_RCV_AND_WAIT_BAD_FILL (for a basic conversation)
The **fill** parameter was set to an invalid value.

000000C1

AP_RCV_IMMD_BAD_STATE
The conversation was not in RECEIVE state.

000000D1

AP_RCV_AND_POST_BAD_STATE
The conversation was not in RECEIVE or SEND state when the TP issued this verb.

000000D2

AP_RCV_AND_POST_NOT_LL_BDY
The conversation was in SEND state; the TP began but did not finish sending a logical record.

000000D5

AP_RCV_AND_POST_BAD_FILL
The **fill** parameter was set to an invalid value.

000000D6

AP_INVALID_SEMAPHORE_HANDLE
The address of the RAM semaphore or system semaphore handle was invalid.

Note APPC cannot trap all invalid semaphore handles. If the TP passes a bad RAM semaphore handle, a protection violation results.

000000D7

AP_BAD_RETURN_STATUS_WITH_DATA
The specified **rtn_status** value was not recognized by APPC.

000000E1

AP_R_T_S_BAD_STATE
The conversation is not in an allowed state when the TP issued this verb.

000000F1

AP_BAD_LL (for a basic conversation)

The logical record length field of a logical record contained an invalid value — 0x0000, 0x0001, 0x8000, or 0x8001. See About Transaction Programs for information on logical records.

000000F2

AP_SEND_DATA_NOT_SEND_STATE
The local TP issued SEND_DATA or MC_SEND_DATA, but the conversation was not in SEND state.

000000F5

AP_SEND_DATA_CONFIRM_ON_SYNC_NONE
The type CONFIRM is not permitted for a conversation that was allocated with a **sync_level** of NONE.

000000F6

AP_SEND_DATA_NOT_LL_BDY (for a basic conversation)
The TP started but did not finish sending a logical record. This occurs only when **type** is one of the following:

AP_SEND_DATA_CONFIRM

AP_SEND_DATA_DEALLOC_FLUSH

AP_SEND_DATA_DEALLOC_SYNC_LEVEL

AP_SEND_DATA_P_TO_R_FLUSH

AP_SEND_DATA_P_TO_R_SYNC_LEVEL

00000102

AP_SEND_ERROR_LOG_LL_WRONG (for a basic conversation)
The LL field of the error log GDS variable did not match the actual length of the data.

00000103

AP_SEND_ERROR_BAD_TYPE (for a basic conversation)
The value of **err_type** was invalid.

00000105

AP_BAD_ERROR_DIRECTION
The specified **err_dir** was not recognized by APPC.

00000150

AP_CNOS_IMPLICIT_PARALLEL
APPC does not permit a program to change the session limit for a mode other than SNASVCMG mode for the implicit partner template when the template specifies parallel sessions. (The term "template" is used because many of the actual values are yet to be filled in).

00000151

AP_CANT_RAISE_LIMITS
APPC does not permit setting session limits to a nonzero value unless the limits currently are zero.

00000152

AP_AUTOACT_EXCEEDS_SESSLIM
On the CNOS verb, the value for **auto_activate** is greater than the value for **partner_lu_mode_session_limit**.

00000153

AP_ALL_MODE_MUST_RESET
APPC does not permit a nonzero session limit when **mode_name_select** indicates ALL.

00000154

AP_BAD_SNASVCMG_LIMITS
Your program specified invalid settings for the **partner_lu_mode_session_limit**, **min_conwinners_source**, or **min_conwinners_target** parameters when **mode_name** was supplied.

00000155

AP_MIN_GT_TOTAL

The sum of **min_conwinners_source** and **min_conwinners_target** specifies a number greater than **partner_lu_mode_session_limit**.

00000156

AP_MODE_CLOSED
The local LU cannot negotiate a nonzero session limit because the local maximum session limit at the partner LU is zero.

00000156

AP_CNOS_MODE_CLOSED
The local LU cannot negotiate a nonzero session limit because the local maximum session limit at the partner LU is zero.

00000157

AP_CNOS_MODE_NAME_REJECT
The partner LU does not recognize the specified mode name.

00000159

AP_RESET_SNA_DRAINS
The SNASVCMG mode does not support the **drain** parameter values.

0000015A

AP_SINGLE_NOT_SRC_RESP
For a single-session CNOS verb, APPC permits only the local (source) LU to be responsible for deactivating sessions.

0000015B

AP_BAD_PARTNER_LU_ALIAS
APPC did not recognize the supplied **partner_lu_alias**.

0000015C

AP_EXCEEDS_MAX_ALLOWED
Your program issued a CNOS verb, specifying a **partner_lu_mode_session_limit** number and **set_negotiable** (NO).

0000015D

AP_CHANGE_SRC_DRAINS
APPC does not permit **mode_name_select** (ONE) and **drain_source** (YES) when **drain_source** (NO) is currently in effect for the specified mode.

0000015E

AP_LU_DETACHED
A command reset the definition of the local LU before the CNOS verb tried to specify the LU.

0000015F

AP_CNOS_COMMAND_RACE_REJECT
The local LU is currently processing a CNOS verb issued by the partner LU.

00000167

AP_SNASVCMG_RESET_NOT_ALLOWED
Your local program attempted to issue the CNOS verbs for the mode named SNASVCMG, specifying a session limit of zero.

000001B4

AP_DISPLAY_INFO_EXCEEDS_LENGTH
The returned DISPLAY information did not fit in the buffer.

000001B5

DISPLAY_INVALID_CONSTANT
The value supplied for NUM_SECTIONS or INIT_SEC_LEN is invalid.

00000506

AP_UNDEFINED_TP_NAME
In the configuration file for your application, APPC could not find an invokable TP name matching the value of **tp_name**.

00000509

AP_ALLOCATE_NOT_PENDING
APPC did not find an incoming allocate (from the invoking TP) to match the value of **tp_name**, supplied by RECEIVE_ALLOCATE. **RECEIVE_ALLOCATE** waited for the incoming allocate and eventually timed out.

00000519

AP_CPSVCMG_MODE_NOT_ALLOWED
The mode named CPSVCMG cannot be specified as the **mode_name** on the deactivate session verb.

00000525

AP_INVALID_PROCESS
The process issuing RECEIVE_ALLOCATE was different from the one started by APPC.

080F6051

AP_SECURITY_NOT_VALID
The user identifier or password specified in the allocation request was not accepted by the partner LU.

084B6031

AP_TRANS_PGM_NOT_AVAIL_RETRY
The remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition may be temporary, such as a time-out. The reason for the error may be logged on the remote node. Retry the allocation.

084C0000

AP_TRANS_PGM_NOT_AVAIL_NO_RETRY
The remote LU rejected the allocation request because it was unable to start the requested partner TP. The condition is permanent. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

10086021

AP_TP_NAME_NOT_RECOGNIZED
The partner LU does not recognize the TP name specified in the allocation request.

10086031

AP_PIP_NOT_ALLOWED
The allocation request specified PIP data, but either the partner TP does not require this data, or the partner LU does not support it.

10086032

AP_PIP_NOT_SPECIFIED_CORRECTLY
The partner TP requires PIP data, but the allocation request specified either no PIP data or an incorrect number of parameters.

10086034

AP_CONVERSATION_TYPE_MISMATCH
The partner LU or TP does not support the conversation type (basic or mapped) specified in the allocation request.

10086041

AP_SYNC_LEVEL_NOT_SUPPORTED
The partner TP does not support the **sync_level** (AP_NONE or AP_CONFIRM_SYNC_LEVEL) specified in the allocation request, or the **sync_level** was not recognized.

Common CSV Return Codes

This section describes the primary and, if applicable, secondary return codes for CSV. The return codes are listed in hexadecimal order.

Primary CSV Return Codes

0000

SV_OK
The verb executed successfully.

0001

SV_PARAMETER_CHECK
The verb did not execute because of a parameter error.

0002

SV_STATE_CHECK
The verb did not execute because it was issued in an invalid state.

0021

SV_CANCELLED
This code is returned for an asynchronous verb when it has been shut down by a WinCSVCleanup call.

0030

SV_FILE_ALREADY_EXISTS
When the SV_NEW file option was used, the file name specified was the name of an existing file.

0031

SV_OUTPUT_DEVICE_FULL
There is insufficient space on the device where the output file resides. Retry the operation after freeing additional disk space.

F006

SV_THREAD_BLOCKING
This verb exceeds the maximum number of simultaneous synchronous verbs allowed.

F008

SV_INVALID_VERB_SEGMENT
The VCB extended beyond the end of the data segment.

F011

SV_UNEXPECTED_DOS_ERROR
One of the following conditions occurred:

- The Microsoft® Windows NT®, Microsoft® Windows® 95, Windows version 3.*x*, Microsoft® MS-DOS®, or OS/2 system encountered an error while processing the verb. The operating system return code was returned through the secondary return code. If the problem persists, contact the system administrator for corrective action.
- A CSV was issued from a message loop that was invoked by another application issuing a Windows environment **SendMessage** function call, rather than the more common Windows environment **PostMessage** function call. Verb processing cannot take place.
- A CSV was issued when **SendMessage** invoked your application. You can determine whether your application has been invoked with **SendMessage** by using the **InSendMessage** Windows API function call.

F012

SV_COMM_SUBSYSTEM_NOT_LOADED
A required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

F024

SV_SERVER_RESOURCE_NOT_FOUND
No communication server was found that could provide the requested function.

F026

SV_SERVER_RESOURCE_LOST

The communications server that was providing the function was lost due to a connection failure.

FFFF

SV_INVALID_VERB
The **opcode** parameter did not match the operation code of any verb. No verb executed.

Secondary CSV Return Codes

00000006

SV_INVALID_DATA_SEGMENT
The data buffer containing the source or target string did not fit in one segment, or the target segment was not a read/write segment. This applies only to the Microsoft® Windows® and OS/2 systems.

00000301

SV_SSCP_PU_SESSION_NOT_ACTIVE
The NMVT was not sent; either the SSCP-PU session was not active, the node configured to receive diagnostic information was not active, or no network management connection was configured.

00000302

SV_DATA_EXCEEDS_RU_SIZE
The data to be sent was too long. The length of the user-supplied data plus headers and added subvectors must fit in a single RU that is not more than 512 bytes long.

00000303

SV_INVALID_DATA_TYPE
The **data_type** parameter contained an invalid value.

00000401

SV_INVALID_DIRECTION
The **direction** parameter contained an invalid value.

00000402

SV_INVALID_CHARACTER_SET
The **char_set** parameter contained an invalid value.

00000404

SV_INVALID_FIRST_CHARACTER
The first character of a type A source string was invalid.

00000405

SV_TABLE_ERROR
One of the following occurred:

- The file containing the user-written type G conversion table was not specified by the environment variable **CSVTBLG**.
- The table was not in the correct format.
- The file specified by the **CSVTBLG** variable was not found.

00000406

SV_CONVERSION_ERROR
One or more characters in the source string were not found in the conversion table. These characters were converted to nulls (0x00). The verb still executed.

00000621

SV_INVALID_MESSAGE_ACTION
The **msg_act** parameter contained an invalid value.

00000624

SV_INVALID_SET
The **dt_set** parameter contained an invalid value.

00000629

SV_COPY_TRACE_IN_PROGRESS
A previously issued COPY_TRACE_TO_FILE is still in progress.

0000062A

SV_TRACE_NOT_STOPPED
A trace was in progress when the verb was issued.

0000062B

SV_INVALID_FILE_OPTION
A value other than SV_NEW or SV_OVERWRITE was specified for **file_option**.

0000062C

SV_TRACE_BUFFER_EMPTY
The trace storage buffer did not contain any data.

0000062F

SV_INVALID_RESET_TRACE
The **reset_trc** parameter contained an invalid value.

00000630

SV_INVALID_CHAR_NOT_FOUND
The **char_not_fnd** parameter contained an invalid value.

00000631

SV_INVALID_SOURCE_CODE_PAGE
The code page specified by **source_cp** is not supported.

00000632

SV_INVALID_TARGET_CODE_PAGE
The code page specified by **target_cp** is not supported.

030000AB

SV_SERVER_COMM_FAILURE
The connection to the server was lost due to physical path problems; for example, the server may have been powered off.

APPC Sample Applications

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about the sample applications that implement the APPC.

This section contains:

- Sample APPC TPs in the SDK

Sample APPC TPs in the SDK

The source code for several sample programs that illustrate using APPC are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\SNA subdirectory on the Host Integration Server 2000 CD-ROM (these samples are located under the \SDK\SAMPLES folder on earlier versions of SNA Server). These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\SNA subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\Sna subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These APPC sample programs include the following:

APPC TP samples	Description
APPC Send and Receive TPs	Sample programs in C that represent simple APPC send and receive transaction programs (TPs) illustrating the use of asynchronous verb completion. These samples are located in the SENDRECV subdirectory implements simple bulk data sending and receiving TPs (SENDTP a nd RECVTP). This sample is located in the \SDK\Samples\SNA\appc subdirectory on the CD-ROM.
Multithreaded Send and Receive TPs	Sample programs in C that represent more advanced APPC send and receive transaction pr ograms illustrating the use of multiple threads and multiple conversations per thread. The m ultithreaded receive TP samples illustrate using events or IO completion ports for notificatio n These samples are located in the \SDK\Samples\SNA\msendrcv subdirectory on the CD-R OM.

In addition to these APPC sample programs, the following supplemental programs are included on the Host Integration Server CD-ROM.

Supplemental program	Description
TPSETUP	A sample installation program in C demonstrating an interface that assists in the configuration of autostarted invokable transaction programs (TPs). This sample is located in the \SDK\Samples\SNA\tpsetup subdirectory on the CD-ROM.
TPSTART	A sample program in C required for the automatic startup of invokable transaction programs that run as applications u nder Microsoft® Windows 2000 and Windows NT®. TPSTART is not required if the transaction program has been writt en as a Windows NT service. TPSTART is also unnecessary under Windows 98 and Windows 95. An executable binary of TPSTART is installed by Host Integration Server Setup in the SYSTEM subdirectory of the Host Integration Server root di rectory. This sample is located in the \SDK\Samples\SNA\tpstart subdirectory on the CD-ROM.

Building the TPs

The APPC samples are designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio .NET interactive development environment (IDE).

To build the APPC samples installed from the Host Integration Server CD-ROM, set the following environment variables:

Variable	Specifies
ISVLIBS	Directory containing the Microsoft® Host Integration Server LIB files for Windows 2000, Windows NT, Windows 98, and Windows 95.
ISVINCS	Directory containing the WINSNA header files
SAMPLEROOT	Root directory of the sample code

For example, if you installed the Host Integration Server SDK directory to the default location (C:\Program Files\Host Integration Server SDK), use the following lines to set the variables (assumes Intel binaries are being produced for Windows 2000, Windows NT on I386, Windows 98, or Windows 95):

```
ISVLIBS=C:\Program Files\Host Integration Server SDK\LIB
ISVINCS=C:\Program Files\Host Integration Server SDK\Include
SAMPLEROOT=C:\Program Files\Host Integration Server SDK\Samples\SNA
```

Change to each subdirectory and run NMAKE on the .MAK file in each directory. For example, for the msendrcv sample, change to the msendrcv subdirectory and type the following:

nmake -f msendrcv.mak

Note that Windows NT on DEC Alpha is not supported by the Host Integration Server SDK. If you wish to build these samples on Windows NT 4.0 for DEC Alpha, the earlier SNA Server 4.0 SDK will be required for accessing the Windows NT import libraries for DEC Alpha under the \SDK\LIB\WINNT\ALPHA folder.

To build the APPC samples installed as part of the MSDN Platform SDK using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory. The default location of this file is C:\Program Files\Microsoft Visual Studio\VC98\Bin (for VS6) or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools (for VS.NET)

To build all the SNA samples, open an MS-DOS Command Prompt window, navigate to the SNA subdirectory, and invoke NMAKE. This will recursively invoke NMAKE and build all of the SNA samples including the APPC samples.

To build a specific sample (SendTP or RecvTP, for example) using the command-line compiler, open an MS-DOS Command Prompt window, navigate to the appropriate subdirectory (SNA\Appc, for example), and invoke NMAKE.

To build a specific sample (SendTp, for example) using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (SNA\appc\sendtp.vcproj, for example) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

TPSETUP

TPSETUP is a program that simplifies the setting of registry or environment variables needed by autostarted invokable TPs. Without an interface like that provided by TPSETUP, configuring of such variables can be complicated and error-prone. Therefore, it is recommended that you use code like TPSETUP in installation programs for autostarted invokable TPs.

Operation

INSTALL.C, the source code for TPSETUP, can be compiled to work in either the Microsoft® Windows 2000, Windows NT®, Windows® 98, Windows® 95 environments or in the Windows version 3.*x* environment. TPSETUP has been constructed so that the program responds correctly in each environment.

For clients running Windows 2000 or Windows NT, it is recommended that autostarted invokable TPs be written as Windows NT services. To create the installation program for such TPs, study the code in INSTALL.C. For example, use the **CreateService** function or similar code when installing a TP that will run as a service under Windows 2000 and Windows NT. (For important information about how services work under Windows 2000 and Windows NT, see the documentation for Windows 2000, Windows NT, and for Win32®.)

TPSTART

An autostarted TP that runs as an application under Microsoft® Windows NT® requires the support of the TPSTART program, which is installed with the Microsoft® Host Integration Server software in the SYSTEM subdirectory of the Host Integration Server root directory. Therefore, the TPSTART program must be started on a Windows 2000-based or Windows NT-based client before an autostarted invokable TP can be started as an application. Starting TPSTART can be accomplished by using standard Windows 2000 and Windows NT methods, such as including TPSTART in the **Startup** group on the client.

APPC Send and Receive TPs

These are simple APPC send and receive TPs that illustrate the use of asynchronous verb completion. This sample implements simple bulk data sending and receiving TPs (SENDTP and RECVTP).

Setup

To set up these TPs, create an appropriate APPC LU-LU-mode triplet. The default is SENDLU-RECVLU-#INTER, but this can be configured (see the following sections).

Input and Output

The APPC send and receive TPs each use a configuration file for input. To name the file, use .CFG as the extension, and use the same base file name as the TP executable file (SENDTP.CFG, for example). Save this configuration file in the same directory location as the executable file (the TP itself).

For SENDTP, the configuration file (called SENDTP.CFG if the executable file is SENDTP.EXE) can contain the following items, one per line, in any order. If a variable is not found in the file or the file is not present at all, the default is used.

Line	Default value	Value to supply
ResultFile =	C:\SENDTP.OUT	File name to print timings to
LocalLUAlias =	SENDLU	Local LU alias
RemoteLUAlias =	RECVLU	Remote LU alias
ModeName =	#INTER	Mode name
LocalTPName =	SENDTP	Name of local TP
RemoteTPName =	RECVTP	Name of remote TP
NumConversations =	1	Number of conversations
NumSends =	2	Number of SEND_DATA verbs per conversation
SendSize =	1024	Size in bytes of data sent each time
ConfirmEvery =	1	Number of **SEND_DATA** verbs between CONFIRM verbs
SendConversation =	No	Yes or No: Use the SEND_CONVERSATION verb rather than the sequence of ALLOCATE, **SEND_DATA**, DEALLOCATE

RECVTP uses a RECVTP.CFG file in a similar way, but only to read the **LocalTPName** field.

The output from SENDTP and RECVTP consists of details of the configuration and the time taken for each conversation, and is sent to the result file specified in SENDTP.CFG.

Operation

RECVTP should be started first; it issues RECEIVE_ALLOCATE with the specified TP name. SENDTP is then started; it first issues MC_SEND_CONVERSATION to tell RECVTP how many conversations will be carried out. It then carries out the specified number of conversations.

For SENDTP, each conversation consists of an MC_ALLOCATE verb, followed by a given number of MC_SEND_DATA verbs of a given size, and interspersed with MC_CONFIRM verbs at a given interval, followed by an MC_DEALLOCATE.

RECVTP issues MC_RECEIVE_AND_WAIT when **RECEIVE_ALLOCATE** completes, and then issues either MC_RECEIVE_AND_WAIT or MC_CONFIRMED according to the return from the previous **MC_RECEIVE_AND_WAIT**.

At any stage, if the TPs encounter an error, they terminate. Use APPC API tracing to diagnose problems with the configuration.

At the end of the specified number of conversations, SENDTP sends timing information to a file.

Both TPs are built from a single source code file, SENDRECV.C. SENDTP is compiled only if **-DSENDTP** is used on the command line.

The TPs run as Microsoft® Windows 2000, Windows NT®, Microsoft® 98, or Windows® 95 applications with a minimized window, the title bar of which displays the status. When the WndProc of this window, TPWndProc, receives the WM_CREATE message for the window, this triggers the issuing of the first verb. When TPWndProc receives an ASYNC_COMPLETE message from Windows APPC, this triggers the issuing of the next verb, dependent on what the previous verb was. When the window is closed, WinAPPCCleanup is issued to terminate any active conversations.

Multithreaded Send and Receive TPs

These multithreaded send and receive TPs are more advanced than the single-threaded equivalents. The samples located in the MSENDRCV subdirectory all use the asynchronous interface of APPC, with verb completion signaled by events (WinAsyncAPPCEx) or IO completion ports (WinAsyncAPPCIOCP). These TPs show how to code multithreaded APPC applications with multiple conversations per thread. They are more complex than the single-threaded equivalents, but are also more realistic.

If you are unfamiliar with APPC, examine the single-threaded TPs first. If you are unfamiliar with methods of creating threads or processing events in Microsoft® Windows 2000, Windows NT®, Windows® 98, and Windows® 95, see the Microsoft® Platform SDK documentation along with the multithreaded TPs.

Setup

There are four multithreaded send and receive routines that illustrate using asynchronous APPC calls:

- MRCV for receiving using events for notification.
- MRCVIO for receiving using IO completion ports for notification.
- MSEND for sending using events for notification
- MSENDRCV for simultaneous sending and receiving using events for notification

To set up these TPs, create an appropriate APPC LU-LU-mode triplet. The default is SENDLU-RECVLU-#INTER, but this can be configured (see the sections that follow). To run a large number of simultaneous conversations, increase the session limits for #INTER or use another mode with large session limits.

One obvious way of configuring these programs is to configure MSEND to run with MRCV or MRCVIO; another way is to configure MSENDRCV to run with another copy of MSENDRCV. However, you can also configure MSEND to run with one or more copies of RECVTP (the single-threaded version) and MRCV or MCRVIO to run with one or more copies of SENDTP. You can also configure MSENDRCV to run with MSEND, MRCV, SENDTP or RECVTP. For more information, see the sections that follow.

One possible arrangement is to place SENDTP (single-threaded) on multiple client computers, and configure MRCV or MCRVIO (multithreaded) on a server so that it interacts with all the TPs on the clients. Many other arrangements are possible.

Configuration for MRCV, MSEND, and MSENDRCV

The MRCV, MSEND, and MSENDRCV TP samples uses a configuration file for configuration and input. To name the file, use .CFG as the extension, and use the same base file name and directory location as the executable file (the TP itself).

The following table shows examples of CFG files that could be used with MSEND and MRCV.

Example of MSEND.CFG file	Example of MRCV.CFG file
ResultFile=MSEND.OUT	TraceFile=MRCV.TRC
TraceFile=MSEND.TRC	LocalTPName=MRCVTP
RemoteTPName=MRCVTP	NumRcvConvs=32
LocalLUAlias=LUA	NumRcvThreads=4
RemoteLUAlias=LUB	RcvSize=4096
ModeName=#INTER	
NumSendConvs=32	
NumSends=128	
ConfirmEvery=16	
SendSize=256	

For MSEND, the configuration file (MSEND.CFG) can contain the following items, one per line, in any order. If a variable is not found in the file or the file is not present at all, the default is used.

Line	Default value	Value to supply
ResultFile =	MSEND.OUT	File name to print timings to (located in default directory for MSEND)
TraceFile =	MSEND.TRC	Trace file name (located in default directory for MSEND)
LocalLUAlias =	SENDLU	Local LU alias
RemoteLUAlias =	RECVLU	Remote LU alias
ModeName =	#INTER	Mode name
RemoteTPName =	MRCVTP	Name of remote TP (for MC_ALLOCATE)
NumSendConvs =	4	Number of conversations to send

NumSends =	8	Number of MC_SEND_DATA verbs per conversation
SendSize =	256	Size in bytes of data sent each time
ConfirmEvery =	2	Number of **MC_SEND_DATA** verbs between MC_CONFIRM verbs

The following lines are for MRCV:

Line	Default value	Value to supply
TraceFile =	MRCV.TRC	Trace file name (located in default directory for MSEND)
LocalTPName =	MRCVTP	Name of local TP (for RECEIVE_ALLOCATE)
NumRcvConvs =	4	Number of conversations to receive
NumRcvThreads =	2	Number of threads to start for processing receive conversations
RcvSize =	4096	Size in bytes of receive buffer for MC_RECEIVE_AND_WAIT

The following table shows examples of configuration files (MSENDRCV.CFG) that could be used with MSENDRCV. Each row of the table (Example A and Example B) contains two files that work together on a pair of computers.

Example A of MSENDRCV.CFG	Example B of MSENDRCV.CFG
ResultFile=MSENDRCV.OUT	ResultFile=MSENDRCV.OUT
TraceFile=MSENDRCV.TRC	TraceFile=MSENDRCV.TRC
LocalTPName=TPA	LocalTPName=TPB
RemoteTPName=TPB	RemoteTPName=TPA
LocalLUAlias=LUA	LocalLUAlias=LUB
RemoteLUAlias=LUB	RemoteLUAlias=LUA
ModeName=#INTER	ModeName=#INTER
NumRcvConvs=50	NumRcvConvs=25
NumRcvThreads=4	NumRcvThreads=4
RcvSize=4096	RcvSize=4096
NumSendConvs=25	NumSendConvs=50
NumSends=100	NumSends=100
ConfirmEvery=10	ConfirmEvery=10
SendSize=256	SendSize=256

The following lines are for MSENDRCV:

Line	Default value	Value to supply
ResultFile =	MSENDRCV.OUT	File name to print timings to (located in default directory for the MSEND or MSENDRCV sending TP)
TraceFile =	MSENDRCV.TRC	Trace file name (located in default directory for the MSEND or MSENDRCV sending TP)
LocalLUAlias =	SENDLU	Local LU alias
RemoteLUAlias =	RECVLU	Remote LU alias
ModeName =	#INTER	Mode name
RemoteTPName =	MRCVTP	Name of remote TP (for MC_ALLOCATE)
NumSendConvs =	4	Number of conversations to send
NumSends =	8	Number of MC_SEND_DATA verbs per conversation
SendSize =	256	Size in bytes of data sent each time
ConfirmEvery =	2	Number of **MC_SEND_DATA** verbs between MC_CONFIRM verbs
LocalTPName =	MRCVTP	Name of local TP (for RECEIVE_ALLOCATE)
NumRcvConvs =	4	Number of conversations to receive
NumRcvThreads =	2	Number of threads to start for processing receive conversations
RcvSize =	4096	Size in bytes of receive buffer for MC_RECEIVE_AND_WAIT

The output from MSEND and MSENDRCV consists of details of the configuration and the time taken for each conversation, and is sent to the result file specified in MSEND.CFG or MSENDRCV.CFG.

Operation of MRCV, MSEND, and MSENDRCV

The MRCV, MSEND, and MSENDRCV TPs use multiple event processing in Windows 2000, Windows NT, Windows 98, or Windows 95 to avoid creating an unnecessary number of threads.

These TPs also use Windows-based processing, but this is incidental. Its only purpose is to display beneath the icon on the screen a running count of threads, the number of conversations currently sending or receiving data, and the number of conversations completed. The Windows-based processing could easily be removed to create a completely batch-oriented program. To do this, termination would need to be signaled with an event rather than with **WM_CLOSE**.

The TP name used in TP_STARTED is the name of the executable file (MSEND, MRCV, or MSENDRCV). The TP names used in MC_ALLOCATE and RECEIVE_ALLOCATE can be configured, as shown in the preceding tables.

MSEND reads its configuration file (or uses defaults) to determine the number of send conversations to start. Each conversation reads the value of NumSends (or uses the default), issues that number of MC_SEND_DATA verbs, and then terminates. When all of the conversations for a thread have terminated, the thread itself terminates. When all of the send threads have terminated, the program terminates.

An MC_CONFIRM verb is issued before the first **MC_SEND_DATA** and then at the intervals specified by **ConfirmEvery**. The complete data flow for a conversation is as follows:

TP_STARTED

MC_ALLOCATE

MC_CONFIRM

MC_SEND_DATA (repeated the number of times specified by **ConfirmEvery**)

MC_CONFIRM

MC_SEND_DATA (repeated the number of times specified by **ConfirmEvery**)

MC_CONFIRM

(Pattern repeats until the number of **MC_SEND_DATA** verbs equals **NumSends**.)

MC_DEALLOCATE

TP_ENDED

MRCV starts up an initial thread for issuing RECEIVE_ALLOCATE verbs, then reads its configuration file (or uses defaults) to determine the number of receive threads to start and the number of conversations to receive.The initial thread issues a **RECEIVE_ALLOCATE** and waits. When the **RECEIVE_ALLOCATE** completes, the initial thread turns the processing of the conversation over to the next available receive thread, and issues another **RECEIVE_ALLOCATE**. This process continues until the configured number of **RECEIVE_ALLOCATE** verbs (that is, **NumRcvConvs**) have completed.

There is a limit to the number of conversations that can be supported on a thread, because of the limit to the number of events that can be waited for with **WaitForMultipleObjects** (a function in the Win32® API). For send threads, the limit is 64 conversations per thread; for receive threads, the limit is 63 conversations per thread.

MSEND works with this limit by starting enough threads to support the configured number of conversations. For example, if **NumSendConvs** is set to 200, four send threads are started: three of them process 64 conversations each and one processes the remaining eight conversations.

MRCV works with this limit by comparing **NumRcvConvs** to **NumRcvThreads**. If **NumRcvConvs** is more than (63 * **NumRcvThreads**), **NumRcvThreads** is increased. If **NumRcvThreads** is greater than **NumRcvConvs**, **NumRcvThreads** is reduced to prevent creating unneeded threads.

With MRCV, to ensure that a receive thread correctly picks up the conversation, two special events are used per thread: *event1* and *event2*. The following table illustrates their use.

RECEIVE_ALLOCATE thread	**Receive thread**
Issue RECEIVE_ALLOCATE and wait	Wait on *event1*
(**RECEIVE_ALLOCATE** completes)	
Select next receive thread and set *event1* for that thread; then wait on *event2* for that thread	
	(*Event1* completes)
	Add conversation to list of conversations being processed
	Set *event2*

(*Event2* completes)	
REPEAT	REPEAT

The receive thread waits not only on the *event1* set for it, but also on one event for each conversation the thread is processing.

If **NumRcvConvs** is set to zero, the **RECEIVE_ALLOCATE** thread will never terminate. If **NumSends** is set to zero, the conversation will never terminate; this is useful for getting the maximum number of simultaneous conversations.

Tracing of MRCV, MSEND, and MSENDRCV

If you want to observe the detailed processing of the MRCV, MSEND, or MSENDRCV sample TPs, you can enable tracing. To do this, find the following line, commented out, near the top of the file:

```
#define SRTRC
```

Enable this line or define this value on the command-line option to the compiler, and trace statements will be written to the trace file(s) specified by the **TraceFile** variable in the configuration.

There are also some trace statements that have been commented out. If they are left commented out, only MC_CONFIRM and MC_CONFIRMED processing is traced while a conversation is running, to maintain a send or receive count without generating a large amount of trace information. You can activate the detailed tracing of events (such as the sending of data) by enabling one or more trace statements.

The **Trace Initiator** (snatrace.exe) tool provides APPC API tracing for Host Integration Server applications (Using the earlier SNA Server, the tracing tool was called **snatrace**). For more information about the **Trace Initiator** and **Trace Viewer** tools, see the Applications and Tools section of the *Microsoft Host Integration Server Guide*.

Configuration for MRCVIO

The MRCVIO TP sample is a multi-threaded console application that uses command-line options for configuration and input. If an option is not provided on the command-line, then the default is used. The following table lists the possible command-line options for MRCVIO.

Command-Line Option	Description
-?	Displays usage information for this sample and exits.
-c numRcvConvs	The maximum number of APPC conversations to support. The default value is 8 with a maximum value of 64.
-d duration	The number of seconds that the sample application should run. The default value is 60 seconds. A value of 0 for duration means run indefinitely.
-h	Displays usage information for this sample and exits.
-i IntTraceFile	The name of the internal trace file if tracing is required. When this command-line option is specified, internal tracing is enabled. This option has no default value and internal tracing is turned off.
-n numRcvThreads	The number of Completion Port threads to allocate. The default value is 4 with a maximum value of 32.
-r rcvSize	The size in bytes of the buffer supplied on each RECEIVE_ALLOCATE. The default value is 4096 with a maximum value of 65535.
-t TPName	The name supplied on the RECEIVE_ALLOCATE verbs. The default value is the "MRCVTP" string.

Operation of MRCVIO

The MRCVIO TP sample uses IO completion ports for notification and will only operate on Windows 2000 or Windows NT. Using the IO completion port mechanism is the preferable method for writing scalable APPC server applications.

The MCRVIO TP sample contains the routines for a multi-threaded console application which uses asynchronous APPC calls on a single I/O completion port to receive data. The MCRVIO sample creates a small pool of threads that will be used for processing

RECEIVE requests. It operates in conjunction with one of the following:

- single-threaded version of send (SENDTP)
- multi-threaded event-based versions of send (MSEND, MSENDRCV)

The MRCVIO sample uses a server model which continues to accept conversations via RECEIVE_ALLOCATE until the application is manually terminated, or a specified timer expires. The conversations do not belong to any particular RECEIVE thread. Each receive thread issues **GetQueuedCompletionStatus** calls to wait for completion of an APPC verb (on any conversation). Each conversation issues **MC_RECEIVE_AND_WAIT** verbs to receive data. If confirmation is requested, an **MC_CONFIRM** verb is issued.

The TP name used in MC_ALLOCATE and RECEIVE_ALLOCATE can be configured using the command-line options as shown in the preceding table of options for the configuration of MCRVIO.

MRCVIO starts up and parses its command-line options (or uses defaults) to determine the number of receive threads to start, the number of conversations to receive, the buffer size for each RECEIVE_ALLOCATE, its TP name, how long the application should run, and whether internal tracing is enabled.

Once command-line options are parsed, the MCRVIO sample calls the **CreateCompletionPort** function to allocate the IO completion port. If this call is successful, then the specified number of threads are created with the thread start routine pointing to the MCRVIO ReceiveThread function and the thread priority for each thread is lowered. Then the specified number of APPC conversations are started.

The MCRVIO sample uses the IO completion port structure and the WinAsyncAPPCIOCP function as listed below.

```
/* IOCP - Structure and function prototype          */
typedef struct
{
    HANDLE       APPC_CompletionPort;
    DWORD        APPC_NumberOfBytesTransferred;
    DWORD        APPC_CompletionKey;
    LPOVERLAPPED APPC_pOverlapped;

} APPC_IOCP_INFO;

extern HANDLE WINAPI WinAsyncAPPCIOCP(
    APPC_IOCP_INFO* iocp_handle, // IO completion port information
    long lpVcb);                 // pointer to APPC verb control block
```

The APPC_CompletionPort must be a HANDLE returned by the **CreateIoCompletionPort** function issued by the application before using **WinAsyncAPPCIOCP**. The other three fields can be set to any value whatsoever. The APPC library does nothing with these other fields, except to return them unaltered on the **GetQueuedCompletionStatus** when the APPC verb completes. An application developer can set these values to whatever they like, but assuming the server application is handling multiple concurrent APPC conversations, an application will need to use one of these three fields to correlate APPC verbs with their completions.

For example, the MCRVIO sample passes a pointer to a Conversation Control Block into the APPC_pOverlapped field when it issues an APPC verb. The same value is returned when the APPC verb completes on the **GetQueuedCompletionStatus**. This allows the sample MCRVIO TP to figure out which APPC verb has actually completed. APPC developers can use a different method (an index into an array of VCBs, for example) to provide the same effect.

Also, an application might use the APPC_CompletionKey field to distinguish between APPC events and other events posted to this IO Completion port. For example, the MCRVIO sample sets this value to a user-defined constant IOCP_VERB_COMPLETE so that the **GetQueuedCompletionStatus** function can distinguish APPC verb completions from the other events that are posted to this IO Completion port (IOCP_START_CONVERSATION, IOCP_END_CONVERSATION and IOCP_TERMINATE_THREAD). However, this is purely for the convenience of the application. An APPC developer could decide not to post any events to its IO Completion port (except implicitly for APPC completions). In such a case, it would be unnecessary to set any value in the APPC_CompletionKey.

CPI-C Applications

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information required to develop C-language applications that use the Common Programming Interface for Communications (CPI-C) to exchange data in a Systems Network Architecture (SNA) environment.

This section contains:

About the CPI-C Guide

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop C-language applications that use the Common Programming Interface for Communications (CPI-C) to exchange data in a Systems Network Architecture (SNA) environment.

This guide is intended for the programmer writing applications that use CPI-C to exchange data. It provides conceptual information and detailed reference information.

To use this guide effectively, you should be familiar with:

- Microsoft® Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows® 2000
    - Microsoft Windows NT®
    - Microsoft Windows 98
    - Microsoft Windows 95
- SNA concepts

This section contains:

Operating Systems Support for CPI-C Development

This section of the guide contains information relating to following operating systems:

- Microsoft® Windows® 2000
- Microsoft Windows NT®
- Microsoft Windows 98
- Microsoft Windows 95
- Microsoft Windows version 3.*x*
- Microsoft MS-DOS®
- OS/2

Microsoft Host Integration Server 2000 supports the development of CPI-C applications for Windows 2000, Windows NT, Windows 98, and Windows 95. Under these operating systems, support for CPI-C applications is provided only for the Win32® subsystem.

The previous Microsoft SNA Server product also supported the development of CPI-C applications for Windows 3.*x* and OS/2. Most CPI-C applications developed for Windows 3.x and OS/2 with SNA Server can be used with Host Integration Server 2000. The Windows 3.x, MS-DOS, and OS/2 interface is described here for completeness, but Windows 3.x, MS-DOS, or OS/2 CPI-C application development is not supported using Host Integration Server.

Finding Further Information on CPI-C

For information about SNA architecture, refer to your system network documentation.

The following documents provide additional information about Host Integration Server application programming interfaces (APIs) based on SNA architecture:

- APPC Applications section of the *Microsoft Host Integration Server Developer's Guide*
- LUA Applications section of the *Microsoft Host Integration Server Developer's Guide*

For more information about SNA and about 3270 information display systems, see the following manuals:

- *IBM 3270 Information Display System: 3274 Control Unit Description and Programmer's Guide*
- *IBM 3270 Information Display System: Color and Programmed Symbols*
- *IBM 3270 Information Display System: 3274 Control Unit Display Station: Operator's Guide*
- *IBM Systems Network Architecture: Technical Overview*
- *IBM Systems Network Architecture: Concepts and Products*
- *IBM Advanced Communications Function Products Installation Guide*
- *IBM Installation and Resource Definition*
- *IBM 9370 LAN Token Ring Support*
- *IBM SNA Format and Protocol Reference Manual: Architectural Logic*

For background information about logical unit (LU) 6.2, Advanced Program-to-Program Communications (APPC), or CPI-C, see the following manuals:

- *IBM Systems Network Architecture: Introduction to APPC*
- *IBM Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2*
- *IBM SNA: Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*
- *IBM SNA: Formats*
- *IBM SNA: Technical Overview*
- IBM SNA: ACF/VTAM Programming for LU Type 6.2

CPI-C Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about developing applications with the Common Programming Interface for Communications (CPI-C).

This section contains:

Introduction to CPI-C

This section introduces the fundamental concepts of the Common Programming Interface for Communications (CPI-C). The following topics are covered:

- Windows CPI-C overview
- Windows CPI-C asynchronous support
- Using asynchronous call completion
- CPI-C call summary
- Conversation characteristics
- Side information
- Configuration
- Operating system considerations

CPI-C is a Systems Application Architecture adherent application programming interface (API) that allows peer-to-peer communications among programs in a Systems Network Architecture (SNA) environment.

Through CPI-C, programs distributed across a network can work together, communicating with each other and exchanging data, to accomplish a single processing task such as querying a remote database, copying a remote file, or sending and receiving electronic mail.

Windows CPI-C Overview

A Windows SNA standard was created to provide one common API to port applications from various operating environments to Microsoft® Windows NT®, Microsoft® Windows® 95, and Microsoft® Windows® version 3.x. As a direct result of this work, Windows CPI-C was developed. The CPI-C calls and information presented in this guide represent an evolving Windows CPI-C that is composed of CPI-C version 1.2 and a set of Windows extensions that allow for multiple applications and asynchronous call completion.

CPI-C version 1.0 was first introduced to provide a means by which two applications could speak and listen to each other; in other words, have a conversation. A conversation is the logical connection between two programs that allows the programs to communicate with each other. Programs using CPI-C converse with each other by making program calls. These calls are used to establish the full characteristics of the conversation, to exchange data, and to control the information flow between the two programs.

CPI-C version 1.1 was extended to include four new areas of function:

- Support for resource recovery (not supported in Windows CPI-C)
- Automatic parameter conversion
- Support for communicating with non-CPI-C programs
- Local/remote transparency

Built upon CPI-C version 1.1, X/Open CPI-C provided:

- Support for nonblocking calls
- The ability to accept multiple conversations
- Support for data conversion (beyond parameters)
- Support for security parameters

CPI-C version 1.2 consolidated CPI-C version 1.1 and X/Open CPI-C by providing all the functions described above. Windows CPI-C adds to this functionality by providing a set of extensions for asynchronous communication in addition to supporting most features in CPI-C version 1.2 with the exception of the following features:

- full duplex operation
- non-blocking call behavior (as defined in the CPI-C 1.2 specification)
- some data conversion functions

For a complete list of unsupported functions, see CPI-C Functions Not Supported.

The use of the Windows CPI-C interface on Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2 will cause additional threads to be created within the calling process. These other threads perform interprocess communication with the SNA Server service over the LAN interface that the client is configured to use (TCP/IP, IPX/SPX, or named pipes, for example).

If an application using Windows CPI-C is running on Windows 2000 or Windows NT, stopping the SNABASE service will cause the application to be unloaded from memory.

Windows CPI-C Asynchronous Support

A program that issues a call and does not regain control until the call completes cannot perform any other operations. This type of operation, referred to as blocking, is not suited to a server application designed to handle multiple requests from many clients. Asynchronous call completion returns the initial call immediately so the application can continue with other processes.

Windows CPI-C support related to asynchronous communications includes the following calls and extensions:

Set_Processing_Mode

Specify_Windows_Handle

Wait_For_Conversation

WinCPICExtractEvent

WinCPICIsBlocking

WinCPICSetBlockingHook

WinCPICSetEvent

WinCPICUnhookBlockingHook

Two methods under Windows 2000, Windows NT, Windows 98, and Windows 95 are available for asynchronous verb completion:

- Message posting using window handles
- Waiting on Win32® events

The traditional method uses messages posted to a window handle to notify an application of verb completion. This method using window handles and messages is also supported on Windows 3.x.

Asynchronous support using message posting is appended to the Set_Processing_Mode call and allows an application to be notified of call completion on a window handle. Calling **RegisterWindowsMessage** with "WinAsyncCPIC" as the string, an application passes a window handle by which the application is notified of call completion. The application then makes the CPI-C call, and when it completes a message is posted to the window handle that was passed, notifying the application that the call is complete.

With the exception of an asynchronous Receive call that can issue certain other calls while pending, a conversation can have only one incomplete operation at any time. For more information on using an asynchronous **Receive** call, see Using Asynchronous Call Completion. In the case of an incomplete operation, the program can issue Wait_For_Conversation to test for its completion or Cancel_Conversation to end the conversation and the incomplete operation.

A second method using Win32 events for notification is supported on Microsoft® Host Integration Server and Microsoft® SNA Server version 3.0 and later. The extension functions using Win32 events (WinCPICSetEvent and WinCPICExtractEvent) will operate only on Windows 2000, Windows NT, Windows 98, and Windows 95. If an event has been registered with the conversation using **WinCPICSetEvent**, then an application can call the Win32 **WaitForSingleObject** or **WaitForMultipleObjects** function to wait to be notified of the completion of the verb.

The only Windows extension functions required for Windows CPI-C are for initialization (WinCPICStartup) and termination (WinCPICCleanup) purposes. Depending on your application, other Windows extensions for handling asynchronous verb completion can be useful, but they are not required. An example of how to use Windows CPI-C asynchronous calls and Windows extensions appears in Using Asynchronous Call Completion. A complete description of all Windows CPI-C calls and extensions appears in CPI-C Calls and Extensions for the Windows Environment.

Before Using Windows CPI-C

The following CPI-C calls and Windows extensions are of particular importance and should be reviewed before using this version of Host Integration Server or earlier versions of SNA Server. Note that the names of the calls are pseudonyms. The actual C function names appear in parentheses after the pseudonym. For example, **Set_Processing_Mode** is the pseudonym for a call. The actual function name is **cmspm**.

Set_Processing_Mode (**cmspm**)
Specifies for the conversation whether subsequent calls will be returned when the operation they request is complete (blocking) or immediately after the operation is initiated (nonblocking). A program is notified of the completion of nonblocking calls when it issues **Wait_For_Conversation** or through a Windows message sent to a WndProc identified by *hwndNotify* in **Specify_Windows_Handle**. When the processing mode is set for a conversation, it applies to all subsequent calls on the conversation until the mode is set again.

Specify_Windows_Handle (**xchwnd**)
Sets the window handle to which a message is sent on completion of an operation in nonblocking mode.

Wait_For_Conversation (**cmwait**)
Waits for the completion of an operation that was initiated when the processing mode conversation characteristic was set to CM_NON_BLOCKING and CM_OPERATION_INCOMPLETE was returned in the *return_code* parameter. Use **Wait_For_Conversation** when running a background thread or a single-threaded application for the Windows 2000, Windows NT, Windows 98, Windows 95, or Windows version 3.*x* systems.

Important An application can set the processing mode by calling **Set_Processing_Mode**. If the window handle is set to NULL, or this call is never issued, then the application must call **Wait_For_Conversation** to be notified when the outstanding operation completes.

When an asynchronous operation is complete, the application's window *hwndNotify* receives the message returned by **RegisterWindowMessage** with "WinAsyncCPIC" as the input string. The *wParam* value contains the conversation return code from the operation that is completing. Its values depend on which operation was originally issued. The *lParam* argument contains the CM_PTR to the conversation identifier specified in the original function call.

WinCPICCleanup
Terminates and deregisters an application from a Windows CPI-C implementation.

> **Important** This function must be called by an application when finished to deregister the application from the Windows CPI-C implementation.

WinCPICExtractEvent
Provides a method for an application to determine the event handle being used for a CPI-C conversation.

WinCPICIsBlocking
Determines if a task is executing while waiting for a previous blocking call to finish. Windows version 3.*x* goes into a **PeekMessageLoop** while allowing Windows to continue. Although a call issued on a blocking function appears to an application as though it blocks, the Windows CPI-C dynamic-link library (DLL) has to relinquish the processor to allow other applications to run. This means that it is possible for the application that issued the blocking call to be re-entered, depending on the message(s) it receives. In this instance, **WinCPICIsBlocking** can be used to determine whether the application task currently has been re-entered while waiting for an outstanding blocking call to finish.

This extension is intended to provide help to an application written to use the CM_BLOCKING characteristic of the Windows **Specify_Processing_Mode** function. **WinCPICIsBlocking** serves the same purpose as **InSendMessage** in the Windows API.

Applications targeted at Windows version 3.*x* that support multiple conversations must specify CM_NONBLOCKING in **Specify_Processing_Mode** so they can support multiple outstanding operations simultaneously. Applications are still limited to one outstanding operation per conversation in all environments. Note that Windows CPI-C prohibits more than one outstanding blocking call per thread.

WinCPICSetBlockingHook
Allows a Windows CPI-C implementation to block CPI-C function calls by means of a new function. Blocking calls apply only if you do not use asynchronous calls. If a function needs to block, the blocking call is called repeatedly until the original request completes. This allows Windows to continue to run while the original application waits for the call to return. Note that while inside the blocking call, the application can be re-entered. **WinCPICSetBlockingHook** is used by Windows version 3.*x* applications that go into a **PeekMessageLoop** to make blocking calls without blocking the rest of the system.

> **Note** By default, Windows 2000, Windows NT, Windows 98, and Windows 95 do not go into a **PeekMessageLoop**; rather, they actually block on an event waiting for the call to complete. The only time you need to use **WinCPICSetBlockingHook** for Windows 2000, Windows NT, Windows 98, and Windows 95 is when a

single-threaded application for Windows 2000, Windows NT, Windows 98, Windows 95, and Windows version 3.x share common source code. In this case, you must explicitly make this call. Contrast this call with **WinCPICIsBlocking** and **WinCPICUnhookBlockingHook**.

WinCPICSetEvent

Associates a Win32 event handle with a verb completion.

WinCPICStartup

Allows an application to specify the version of Windows CPI-C required and to retrieve details of the specific CPI-C implementation.

An application must call this function to register itself with a Windows CPI-C implementation before issuing any further Windows CPI-C calls.

WinCPICUnhookBlockingHook

Removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

Using Asynchronous Call Completion

With one exception, Host Integration Server and the earlier SNA Server permit one outstanding Windows SNA asynchronous call per connection and one blocking verb per thread. The exception to this guideline is that when issuing an asynchronous Receive call, the following calls can be issued while the **Receive** is outstanding:

Cancel_Conversation

Deallocate

Request_To_Send

Send_Error

Test_Request_To_Send_Received

This allows an application, in particular a 5250 emulator, to use an asynchronous **Receive** to receive data. Use of this feature is strongly recommended.

The following example illustrates how to use asynchronous call completion with Host Integration Server or SNA Server:

```
void ProcessVerbCompletion (WPARAM wParam LPARAM lParam)

{
    for ( i = 0; i<nPendingVerbs; i++ )
        if (memcmp  (pPending [i].ConvID, (Conversation_ID) lParam)== 0)
            ProcessCommand  (wParam, lParam);
}

LRESULT CALLBACK SampleWndProc  ( . . .)
{
    if (msg = = uAsyncCPIC )   {
        ProcessVerbCompletion  (wParam, lParam);
    }
    else switch  (msg)  {
       case WM_USER:
        Initialize_Conversation  ( lpConvId,  "GORDM",  &lError  );
        if  ( lError  ! = CM_OK )   {
            ErrorDisplay ( ) ;
            break ;
        }
        Set_Processing_Mode  ( lpConvId,  CM_NON_BLOCKING,  &lError  ) ;
        if  ( lError ! = CM_OK )  {
            ErrorDisplay ( ) ;
            break ;
        }
        Allocate  ( lpConvId,  &lError )  ;
        switch  ( lError )  {

        case CM_OK:
            break ;

        case CM_OPERATION_INCOMPLETE:
            memcopy  (pPending [nPending ++].ConvId,  lpConvId,  sizeof  (C) ;
            break ;
        default:
            ErrorDisplay ( ) ;

        }
        break ;
}

WinMain  ( . . . )
{
    if  ( ( WinCPICStartup  ( . . . )  =  =  FALSE  )   {
        return FALSE;

    }
```

```
      uAsyncCPIC = RegisterWindowMessage  ("WinAsyncCPIC"");
      Specify_Windows_Handle  (hwndSample) ;
      while  (GetMessage  ( . . . )   )   {
      . . . . .
      }
      WinCPICCleanup  ( . . . )

  }
```

For more information on CPI-C calls and Windows extensions, see CPI-C Calls and Extensions for the Windows Environment. For additional information on using CPI-C, see the *IBM Systems Application Architecture Common Programming Interface Communications Reference*, part number SC26-4399-04.

CPI-C Call Summary

This section briefly describes each CPI-C call. The features provided by a CPI-C call can be broader than this summary indicates. The calls are grouped in categories according to the function they perform. There are calls that start and stop a conversation, send and receive data, get information, and get side information.

> **Note** The names of the calls are pseudonyms. The actual C function names appear in parentheses after the pseudonyms. For example, **Initialize_Conversation** is the pseudonym for a call. The actual function name is **cminit**.

Starting a Conversation

The calls in this category are used to start a conversation between two programs.

Accept_Conversation (**cmaccp**)
Issued by the invoked program to accept the incoming conversation and set certain conversation characteristics. Upon successful execution of this call, CPI-C generates a conversation identifier.

Allocate (**cmallc**)
Issued by the invoking program to allocate a conversation with the partner program, using the current conversation characteristics. CPI-C can also start a session between the local LU and partner LU if one does not already exist. The type of conversation allocated depends on the conversation type characteristic—mapped or basic.

Initialize_Conversation (**cminit**)
Issued by the invoking program to obtain a conversation identifier and to set the initial values for the conversation's characteristics. The initial values are derived from side information associated with the symbolic destination name or are CPI-C defaults.

After issuing **Initialize_Conversation**, the invoking program can issue any of the following **Set_** calls to change the initial conversation characteristics. These calls cannot be issued after **Allocate** has been issued.

Call	**Sets**
Set_Conversation_Security_Password (**cmscsp**)	Security password
Set_Conversation_Security_Type (**cmscst**)	Conversation security type
Set_Conversation_Security_User_ID (**cmscsu**)	Security user identifier
Set_Conversation_Type (**cmsct**)	Conversation type
Set_Mode_Name (**cmsmn**)	Mode name
Set_Partner_LU_Name (**cmspln**)	Partner LU name
Set_Return_Control (**cmsrc**)	Return control
Set_Sync_Level (**cmssl**)	Synchronization level
Set_TP_Name (**cmstpn**)	Program name

Sending Data

The following calls are used to send data to the partner program.

Confirm (**cmcfm**)
Sends the contents of the local LU's send buffer and a confirmation request to the partner program and waits for confirmation.

Flush (**cmflus**)
Sends the contents of the local LU's send buffer to the partner LU (and partner program). If the send buffer is empty, no action takes place.

Prepare_To_Receive (**cmptr**)
Changes the state of the conversation for the local program from SEND to RECEIVE, making it possible for the local program to begin receiving data. Before changing the conversation state, this call performs the equivalent of the **Flush** or **Confirm** call.

Request_To_Send (**cmrts**)
Notifies the partner program that the local program wants to send data. The partner program may or may not act on this request.

Send_Data (**cmsend**)
Puts data in the local LU's send buffer for transmission to the partner program. The data collected in the local LU's send buffer is transmitted to the partner LU (and partner program) when one of the following occurs:

- The send buffer fills up.
- The local program issues a **Flush**, **Confirm**, or Deallocate call or other call that flushes the LU's send buffer. (Some send types, set by **Set_Send_Type**, include flush functionality.)

Set_Prepare_To_Receive_Type (**cmsptr**)
Sets the conversation's prepare-to-receive type, which specifies whether subsequent **Prepare_To_Receive** calls will include **Flush** or **Confirm** functionality. The prepare-to-receive type affects all subsequent **Prepare_To_Receive** calls. It can be changed by reissuing **Set_Prepare_To_Receive_Type**.

Set_Send_Type (**cmsst**)
Sets the conversation's send type. The send type specifies how data will be sent by **Send_Data**. The send type can specify that only data be sent or that, in addition to sending data, CPI-C execute the equivalent of **Flush**, **Confirm**, **Prepare_To_Receive**, or Deallocate. The send type value affects all subsequent **Send_Data** calls. It can be changed by reissuing **Set_Send_Type**.

Receiving Data

The following calls or extensions allow a program to receive data from its partner program.

Receive (**cmrcv**)
: Issuing this call while the conversation is in RECEIVE state causes the local program to receive any data that is currently available from the partner program. If no data is available and the receive type is set to CM_RECEIVE _AND_WAIT, the local program waits for data to arrive. If the receive type is set to CM_RECEIVE_IMMEDIATE, the program does not wait.

  Issuing this call while the conversation is in SEND or SEND_PENDING state is allowed only if the receive type is set to CM_RECEIVE_AND_WAIT. This flushes the LU's send buffer and changes the conversation state to RECEIVE. The local program then begins to receive data.

Set_Fill (**cmsf**)
: Used in a basic conversation, this call sets the conversation's fill type, which specifies whether programs will receive data in the form of logical records or as a specified length of data. This call has an effect only in basic conversations. The fill value affects all subsequent **Receive** calls. It can be changed by reissuing **Set_Fill**.

Set_Processing_Mode (**cmspm**)
: Specifies for the conversation whether subsequent calls will be returned when the operation they have requested is complete (blocking) or immediately after the operation is initiated (nonblocking). A program is notified of the completion of nonblocking calls when it issues Wait_For_Conversation or through a Windows message sent to a WndProc identified by the *hwndNotify* parameter in **Specify_Windows_Handle**.

Set_Receive_Type (**cmsrt**)
: Sets the conversation's receive type, which specifies whether a program issuing a **Receive** call will wait for data to arrive if data is not available. The receive type value affects all subsequent **Receive** calls. It can be changed by reissuing **Set_Receive_Type**.

Specify_Windows_Handle (**xchwnd**)
: Sets the window handle to which a message is sent on completion of an operation in nonblocking mode. An application can set the processing mode by calling **Set_Processing_Mode**. If the window handle is set to NULL or this call is never issued, then the application must call **Wait_For_Conversation** to be notified when the outstanding operation completes.

WinCPICSetBlockingHook
: Allows a Windows CPI-C implementation to block CPI-C function calls by means of a new function. This call must be explicitly issued for Windows version 3.*x* applications to make blocking calls without blocking the rest of the system.

  A Windows CPI-C implementation has a default mechanism by which blocking CPI-C functions are implemented. This function enables the application to execute its own function at blocking time in place of the default function.

Confirming Receipt of Data and Reporting Errors

The following calls confirm receipt of data or report an error.

Confirmed (**cmcfmd**)
: Replies to a confirmation request from the partner program. It informs the partner program that the local program has not detected an error in the received data. Because the program issuing the confirmation request waits for a confirmation, **Confirmed** synchronizes the processing of the two programs.

Send_Error (**cmserr**)
: Notifies the partner program that the local program has encountered an application-level error. The local program can use **Send_Error** to inform the partner program of an error encountered in received data, to reject a confirmation request, or to truncate an incomplete logical record it is sending.

Set_Error_Direction (**cmsed**)
: Specifies whether a program detected an error while receiving data or while preparing to send data. Error direction is relevant only when a program issues **Send_Error** in SEND_PENDING state—immediately after issuing Receive and receiving data as well as a *status_received* value of CM_SEND_RECEIVED.

Set_Log_Data (**cmsld**)
: Used in a basic conversation, this call specifies a log message (log data) and its length to be sent to the partner LU. This call has an effect only in basic conversations. If present, log data is sent when **Send_Error** is issued or when the conversation is abnormally deallocated. After the log data is sent, CPI-C resets the log data to NULL and the log data length to zero.

Getting Information

The following calls retrieve information about the characteristics of a specified conversation.

Extract_Conversation_Security_Type (**xcecst**)
Retrieves security type.
Extract_Conversation_Security_User_ID (**cmecsu**)
Retrieves security user identifier.
Extract_Conversation_State (**cmecs**)
Retrieves conversation state.
Extract_Conversation_Type (**cmect**)
Retrieves conversation type.
Extract_Mode_Name (**cmemn**)
Retrieves mode name.
Extract_Partner_LU_Name (**cmepln**)
Retrieves partner LU name.
Extract_Sync_Level (**cmesl**)
Retrieves synchronization level.
Test_Request_To_Send_Received (**cmtrts**)
Determines whether a request-to-send notification has been received from the partner program.

Ending a Conversation

The following calls end a conversation.

Deallocate (**cmdeal**)
: Deallocates a conversation between two programs. Before deallocating the conversation, this call performs the equivalent of the Flush or Confirm call, depending on the current conversation synchronization level and deallocate type.

Set_Deallocate_Type (**cmsdt**)
: Specifies how the conversation is to be deallocated. The deallocation instructions specified by this call take effect when **Deallocate** is issued or when the send type is set to CM_SEND_AND_DEALLOCATE and Send_Data is issued.

Administering Side Information

The following calls let CPI-C applications add, replace, retrieve, or delete side information entries from memory. (See Side Information.)

Delete_CPIC_Side_Information **(xcmdsi)**
Deletes side information entry.

Extract_CPIC_Side_Information **(xcmesi)**
Retrieves side information.

Set_CPIC_Side_Information **(xcmssi)**
Adds or replaces side information entry.

Initial Conversation Characteristics

CPI-C maintains a set of internal values called characteristics for each conversation. Some characteristics affect the overall operation of the conversation, such as the conversation type. Others affect the behavior of specific calls, such as the receive type.

Many of these characteristics are initially derived from the side information table (see Side Information) in memory. Initialize_Conversation specifies the symbolic destination name (*sym_dest_name*) associated with the desired side information table entry.

The following table lists the initial values of the conversation characteristics and tells which call can change a given value.

Characteristic	Initial value set by Initialize_Conversation	Initial value set by Accept_Conversation	Can be changed by
Conversation state	CM_INITIALIZE_STATE	CM_RECEIVE_STATE	Depends on call
Conversation type	CM_MAPPED_ CONVERSATION	The value specified by the invoking program.	Set_Conversation_ Type
Deallocate type	CM_DEALLOCATE_ SYNC_LEVEL	CM_DEALLOCATE_ SYNC_LEVEL	Set_Deallocate_ Type
Error direction	CM_RECEIVE_ERROR	CM_RECEIVE_ ERROR	Set_Error_Direction
Fill	CM_FILL_LL	CM_FILL_LL	Set_Fill
Log data	Null	Null	Set_Log_Data
Log data length	0	0	Set_Log_Data
Mode name	The mode name contained in the side information. If no *sym_dest_name* is specified, this is a null string.	The mode name for the session on which the conversation startup request arrived.	Set_Mode_Name
Mode name length	Length of mode name. If no *sym_dest_name* is specified, this is zero.	Length of mode name.	Set_Mode_Name
Partner LU name	The partner LU name contained in the side information. If no *sym_dest_name* is specified, this is a single blank.	The partner LU name for the session on which the conversation startup request arrived.	Set_Partner_LU_ Name
Partner LU name length	Length of partner LU name. If no *sym_dest_name* is specified, this is 1.	Length of partner LU name.	Set_Partner_LU_ Name
Partner program name	The program name contained in the side information. If no *sym_dest_name* is specified, this is a single blank.	Not applicable.	Set_TP_Name
Partner program name length	Length of partner program name. If no *sym_dest_name* is specified, this is 1.	Not applicable.	Set_TP_Name
Password	The password contained in the side information. If no *sym_dest_name* is specified, this is a single blank.	The value specified by the invoking program.	Set_Conversation_ Security_Password
Password length	Length of password. If no *sym_dest_name* is specified, this is 1.	Length of password.	Set_Conversation_ Security_Password
Prepare-to-receive type	CM_PREP_TO_ RECEIVE_SYNC_ LEVEL	CM_PREP_TO_ RECEIVE_SYNC_ LEVEL	Set_Prepare_To_ Receive_Type
Receive type	CM_RECEIVE_AND_ WAIT	CM_RECEIVE_AND_ WAIT	Set_Receive_Type
Return control	CM_WHEN_SESSION_ ALLOCATED	Not applicable.	Set_Return_Control
Security type	The security type contained in the side information.	The value specified by the invoking program.	Set_Conversation_ Security_Type
Send type	CM_BUFFER_DATA	CM_BUFFER_DATA	Set_Send_Type
Synchronization level	CM_NONE	The value specified by the invoking program.	Set_Sync_Level
User identifier	The user identifier contained in the side information. If no *sym_dest_name* is specified, this is a single blank.	The value specified by the invoking program.	Set_Conversation_ Security_User_ID

User identifier length	Length of user identifier. If no *sym_dest_name* is specified, this is 1.	Length of user identifier.	Set_Conversation_Security_User_ID

Side Information

The information required for two CPI-C programs to communicate is stored as a table, called the side information table, in memory. The table is derived from the symbolic destination name (configured in Host Integration Server or SNA Server) and from the Set_CPIC_Side_Information, Extract_CPIC_Side_Information, and Delete_CPIC_Side_Information calls.

The side information is maintained by the system administrator. For additional information about configuration, see the Administrator's Reference section of the *Microsoft Host Integration Server Guide* or the *Microsoft SNA Server Administration Guide* provided with earlier versions of SNA Server.

If you are developing commercial programs or programs that will be installed on multiple computers within your organization, it is recommended that you include logic that allows a user or system administrator to specify configuration information for each copy of the program.

Each side information entry contains the following fields:

Symbolic destination name
This is the *sym_dest_name* parameter specified by Initialize_Conversation. It is the identifier for the side information entry. The name can be up to eight ASCII characters. See Set_CPIC_Side_Information for the allowed characters.

Partner LU name
This is the name by which the partner LU is known to the local program. It can be an alias of up to eight ASCII characters or a fully qualified network name of up to 17 characters. See Set_Partner_LU_Name for the allowed characters.

Partner program type and name
These fields indicate whether the partner program is an application transaction program (TP) or an SNA service TP, and provide the partner program name. An application TP name can contain up to 64 ASCII characters. A service TP name can contain up to four characters. See Set_TP_Name for the allowed characters.

Mode name
This name represents a set of characteristics to be used in an LU-to-LU session. The mode name can contain up to eight ASCII characters. See Set_Mode_Name for the allowed characters.

Conversation security type
This field indicates whether security will be used and if so, what type.

You can use conversation security to require that the invoking program provide a user identifier and password before CPI-C will allocate a conversation with the invoked program.

For an invoked program that in turn invokes another program, the security type can inform the second invoked program that security has already been verified.

See Set_Conversation_Security_Type for further information about conversation security.

Security user identifier and password
If you intend to use conversation security, a valid combination of user identifier and password is required to access the invoked program. The user identifier and password can be up to 10 ASCII characters. See Set_Conversation_Security_User_ID and Set_Conversation_Security_Password for the allowed characters.

Configuration

In addition to maintaining the side information (specified by *sym_dest_name*), the system administrator must define the following entities during configuration:

- Modes
- Local logical units (LUs)
- Partner LUs
- Invokable programs
- User identifiers and passwords

Note For a user or group using TPs, 5250 emulators, or Advanced Program-to-Program Communications (APPC) applications, you can assign a default local APPC LU and a default remote APPC LU. These default LUs are accessed when the user or group member starts an APPC program (a TP, 5250 emulator, or APPC application) and the program does not specify LU aliases by leaving the field NULL or filling with blanks. For more information, see the Administrator's Reference section of the Microsoft Host Integration Server 2000 Guide or the Microsoft SNA Server Administration Guide provided with earlier versions of SNA Server.

Windows 2000, Windows NT, Windows 98, and Windows 95 Considerations

This topic summarizes things to keep in mind when you are developing programs on a server based on Microsoft® Windows 2000, Microsoft Windows NT®, Microsoft Windows 98, or Microsoft Windows 95.

Host Integration Server 2000 with Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

Asynchronous completion notification using message posting

When an asynchronous operation is complete, the application's window *hwndNotify* receives the message returned by **RegisterWindowMessage** with "WinAsyncCPIC" as the input string. The *wParam* value contains the *conversation_return_code* from the operation that is completing. Its values depend on which operation was originally issued. The *lParam* argument contains the CM_PTR to the *conversation_ID* specified in the original function call.

Asynchronous completion notification using Win32 events

When a verb is issued on a non-blocking conversation, it returns CM_OPERATION_INCOMPLETE if it is going to complete asynchronously. If an event has been registered with the conversation, then the application can call **WaitForSingleObject** or **WaitForMultipleObjects** to be notified of the completion of the verb. WinCPICExtractEvent allows a CPI-C application to determine this event handle. After the verb has completed, the application must call Wait_for_Conversation to determine the return code for the asynchronous verb. The Cancel_Conversation function can be called to cancel an operation and the conversation itself.

It is the responsibility of the application to reset the event, as it is with other APIs.

If no event has been registered, then the asynchronous verb completes as it does at present, which is by posting a message to the window that the application has registered with the CPI-C library.

Byte ordering

By default, Intel byte ordering is used. For inline environments, defining NON_INTEL_BYTE_ORDER will do all the required flipping for constants. Nonconstant input parameters in verb control blocks (VCBs)—for example, lengths and pointers—are always in the native format.

Events

To receive data asynchronously, an event handle is passed in the semaphore field of the VCB. This event must be in the nonsignaled state when passed to CPI-C, and the handle must have EVENT_MODIFY_STATE access to the event.

Library name

In preparation for the coexistence of Win16 and Win32® API libraries on the same computer, the Win32 DLL name has been changed from WINCPIC.DLL to WINCPIC32.DLL.

The old DLL name should be used for Win32-based applications that are required to run on Microsoft® SNA Server version 2.0. The new DLL name should be used for Win32-based applications that are intended to run only on Microsoft® Host Integration Server or on SNA Server version 2.1 or later versions.

If you intend your Win32-based application to be used with SNA Server version 2.0, you should link with the library included with SNA Server version 2.0. Otherwise use the library provided with Host Integration Server.

Multiple threads

A TP can have multiple threads that issue verbs. Windows CPI-C makes provisions for multithreaded Windows-based processes. A process contains one or more threads of execution. All references to threads refer to actual threads in a multithreaded Windows environment.

Packing

For performance reasons, the VCBs are not packed. As a result of this, DWORDs are on DWORD boundaries, WORDs on WORDs, and BYTEs on BYTEs. VCBs should be accessed using the structures provided.

Run-time linking

For a TP to be dynamically linked to CPI-C at run time, the TP must issue:

- **LoadLibrary** to dynamically load WINCPIC.DLL or WINCPIC32.DLL, the libraries for WINCPIC.
- **GetProcAddress** to specify WINCPIC as the desired entry point to the dynamic-link library (DLL).

Issue the **FreeLibrary** call when the CPI-C library is no longer required.

Simultaneous conversations

A program can simultaneously participate in as many as 64 conversations per process.

Terminating applications

In the Windows 2000, Windows NT, Windows 98, and Windows 95 environments, CPI-C cannot tell when an application terminates. Therefore, if an application must close (for example, it receives a WM_CLOSE message as a result of an ALT+F4 from a user), the application should call WinCPICCleanup.

Yielding to other components

When processing CPI-C and Common Service Verbs (CSV), it may be necessary for the library code to yield to allow another component, such as the SnaBase, to receive messages and pass them to the application. This can be accomplished by using the Windows extensions WinCPICSetBlockingHook and WinCPICUnhookBlockingHook.

WinCPICSetBlockingHook allows a Windows CPI-C implementation to block CPI-C function calls by means of a new function. This call is used by Windows version 3.*x* applications to make blocking calls without blocking the rest of the system. To call **WinCPICSetBlockingHook**:

```
FARPROC WINAPI WinCPICSetBlockingHook (FARPROC lpBlockFunc)
```

WinCPICUnhookBlockingHook removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism. To call **WinCPICUnhookBlockingHook**:

```
BOOL WINAPI WinCPICUnhookBlockingHook (void)
```

Windows 3.x Considerations

This topic summarizes things to keep in mind when you are developing programs on Microsoft® Windows® 3.x.

Asynchronous completion notification

When an asynchronous operation is complete, the application's window *hwndNotify* receives the message returned by **RegisterWindowMessage** with "WinAsyncCPIC" as the input string. The *wParam* value contains the *conversation_return_code* from the operation that is completing. Its values will depend on which operation was originally issued. The *lParam* argument contains the CM_PTR to the *conversation_ID* specified in the original function call.

Load-time linking

For a program to be dynamically linked to CPI-C at load time, you must do one of the following at link time:

- Insert the following **IMPORTS** statement in the definition (.DEF) file used to link the program:

  IMPORTS WINCPIC.[*entry point to be used*]

  (Use this statement for each entry point needed.)

- Link the program to WINCPIC.LIB, which contains the entry-point linkage information for CPI-C. If you intend to use common service verbs (CSV), you must also link to WINCSV.LIB, which contains the entry point information for CSV.

Local LUs

CPI-C does not provide a parameter for a program to specify the local LU it wants to use. The APPCLLU environment variable specifies a local LU. This variable can be set through the application section of the WIN.INI file, as in the following example:

```
[Application]
APPCTPN=TP1
APPCLLU=LU1
```

Setting APPCLLU is only necessary if the program does not use an LU from the default LU pool.

Packing

VCBs are not packed. As a result, DWORDs and WORDs are on WORD boundaries, and BYTEs are on BYTE boundaries. This means, for example, that there is not a 2-byte gap between the primary and secondary return codes. VCBs should be accessed using the structures provided, and compiler options that change this packing method should be avoided.

Run-time linking

For a program to be dynamically linked to CPI-C at run time, the program must issue:

- **LoadLibrary** to dynamically load WINCPIC.DLL, the CPI-C library.
- **GetProcAddress** to specify CPI-C as the desired entry point to the DLL.

Issue the **FreeLibrary** call when the CPI-C library is no longer required.

Simultaneous conversations

A program can participate in as many as 64 conversations simultaneously with the Windows environment. However, if more than one CPI-C application is active at once, the total number of conversations cannot exceed 64.

Terminating applications

In the Windows environment, CPI-C cannot tell when an application terminates. Therefore, if an application must close (for example, it receives a WM_CLOSE message as a result of an ALT+F4 from a user), the application should call WinCPICCleanup.

TP names

When a program issues Initialize_Conversation or Accept_Conversation, SNA Server generates an instance of a TP.

CPI-C does not provide a parameter for specifying the name of the invoking (**Initialize_Conversation**) TP instance. Instead, it is provided by setting the APPCTPN variable in the application section of the WIN.INI file, as in the following example:

```
[MyApplication]
APPCTPN=TP1
```

If APPCTPN is not set, the default value is CPIC_DEFAULT_TPNAME.

For the invoked program, the value of APPCTPN must match the value set by the invoking program. **Accept_Conversation** cannot complete unless the allocation request from the invoking program specifies the TP name contained in APPCTPN. The invoked program also sets APPCTPN through an application section of the WIN.INI file.

If this variable is not set when the invoked program issues **Accept_Conversation**, the default value is CPIC_DEFAULT_TP_NAME.

The APPCTPN variable can be an ASCII string from 1 through 64 characters long, consisting of uppercase and lowercase letters, numerals 0 through 9, and special characters, except the space. The APPCTPN variable cannot be set to an SNA service TP name, which contains nonprintable hexadecimal values.

If the invoking program issues multiple **Initialize_Conversation** calls, it can set APPCTPN to a different value before each call.

Yielding to other components

When processing CPI-C and CSV, it may be necessary for the library code to yield to allow another component, such as the SnaBase, to receive messages and pass them to the application. This can be accomplished by using WinCPICSetBlockingHook and WinCPICUnhookBlockingHook.

WinCPICSetBlockingHook allows a Windows CPI-C implementation to block CPI-C function calls by means of a new function. This call is used by Windows version 3.*x* applications to make blocking calls without blocking the rest of the system. To call **WinCPICSetBlockingHook**:

```
FARPROC WINAPI WinCPICSetBlockingHook (FARPROC lpBlockFunc)
```

WinCPICUnhookBlockingHook removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism. To call **WinCPICUnhookBlockingHook**:

```
BOOL WINAPI WinCPICUnhookBlockingHook (void)
```

OS/2 Considerations

This topic summarizes processing considerations you need to be aware of when developing programs on an OS/2 server, client, or workstation.

Critical sections

Exercise great caution when using critical sections, which are the parts of a program that must run without interruption. A program must not issue a CPI-C call within a critical section.

Load-time linking

For a program to be dynamically linked to CPI-C at load time, you must do one of the following at link time:

- Insert the following **IMPORTS** statement in the definition (.DEF) file used to link the program:

  IMPORTS CPIC.[*entry point to be used*]

  (Use this statement for each entry point needed.)

- Link the program to WINCPIC.LIB, which contains the entry-point linkage information for CPI-C.

Local LUs

CPI-C does not provide a parameter for a program to specify the local LU it wants to use. The APPCLLU environment variable specifies a local LU. This variable can be set:

- By the program itself.
- By the operator if the program is operator-started.
- During configuration if the program is automatically started.

Setting APPCLLU is necessary only if the program does not use an LU from the default LU pool.

Multiple processes

Multiple processes cannot have the same conversation identifier. Only the process that issues Initialize_Conversation or Accept_Conversation can use the conversation identifier returned by the call. Another process wanting to use CPI-C must issue **Initialize_Conversation** or **Accept_Conversation** to obtain its own conversation identifier.

Two or more instances of the same program can be run as different processes.

One process can engage in multiple conversations, subject to the restrictions described under "Simultaneous conversations" in this topic.

Multiple threads

A program can have multiple threads that issue calls. However, the program cannot issue two calls simultaneously on the same conversation. If CPI-C is issuing a call and another thread of the program issues a call on the same conversation, the thread will hang until the first call completes.

OS/2 exception TRAP 000D

The OS/2 exception TRAP 000D occurs when CPI-C is unable to pass a return code to the local program because the return code pointer parameter supplied to CPI-C is invalid.

Packing

VCBs are not packed. As a result, DWORDs and WORDs are on WORD boundaries, and BYTEs are on BYTE boundaries. This means, for example, that there is not a 2-byte gap between the primary and secondary return codes. VCBs should be accessed using the structures provided, and compiler options that change this packing method should be avoided.

Run-time linking

For a program to be dynamically linked to CPI-C at run time, the program must issue:

- **DosLoadModule** to dynamically load CPIC.DLL, the CPI-C library.
- **DosGetProcAddr** to specify the desired entry points to the DLL. Each CPI-C call is an entry point to the DLL.

Unlinking (the **DosFreeModule** call) is not supported.

Simultaneous conversations

A program can simultaneously participate in as many as 64 conversations for each OS/2 process.

Stack size

The recommended stack size for a program is at least 3000 bytes.

When executing a call, CPI-C uses the calling program's stack. The combination of OS/2 and CPI-C requires 2560 bytes of stack space, and the program requires additional stack space for its variables.

Terminating applications

When an application terminates, it should issue the APPC TP_ENDED verb with the type set to AP_HARD for all active TPs.

TP names

When a program issues Initialize_Conversation or Accept_Conversation, SNA Server generates an instance of a TP.

CPI-C does not provide a parameter for specifying the name of the invoking (**Initialize_Conversation**) TP instance. Instead, it is provided through the APPCTPN environment variable. For the invoking program, APPCTPN can be set by the operator or by the program itself. If APPCTPN is not set, the default value is CPIC_DEFAULT_TPNAME.

For the invoked program, the value of APPCTPN must match the value set by the invoking program. **Accept_Conversation** cannot be completed unless the allocation request from the invoking program specifies the TP name contained in APPCTPN.

If the invoked program is operator-started, the value of APPCTPN can be set by the operator or by the program. If the program is automatically started, the value of APPCTPN is set when configuring the invokable program. It can also be set by the program itself. If this variable is not set when the invoked program issues **Accept_Conversation**, the default value is CPIC_DEFAULT_TP_NAME.

The APPCTPN variable can be an ASCII string from 1 through 64 characters long, consisting of uppercase and lowercase letters, numerals from 0 through 9, and special characters, except the space. The APPCTPN variable cannot be set to an SNA service TP name, which contains nonprintable hexadecimal values.

If the invoking program issues multiple **Initialize_Conversation** calls, it can set APPCTPN to a different value before each call.

Writing CPI-C Applications

A processing task accomplished by programs using CPI-C is called a transaction. Consequently, programs that use CPI-C are called transaction programs, or TPs. These programs communicate as peers, on an equal (rather than hierarchical) basis. The TPs use CPI-C calls to exchange status information and application data. Each TP uses CPI-C calls to supply parameters to CPI-C, which performs the desired function and returns parameters to the TP.

TPs distributed across a local or wide area network perform distributed transaction processing.

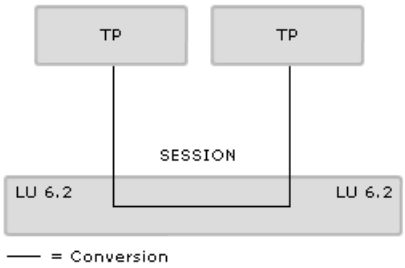
This section describes how to write transaction programs using CPI-C and how to configure the systems on which TPs run. The topics in this section cover the following general areas:

- Understanding fundamental concepts related to TPs
- Designing and coding TPs
- Configuring registry and environment variables for invokable TPs
- Configuring Microsoft® Host Integration Server or the earlier Microsoft® SNA Server to work with your TPs

This section contains:

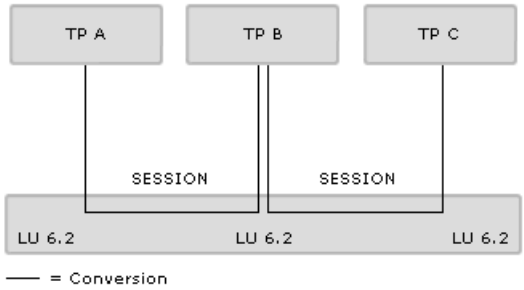
Communication Between TPs

Various hardware and software elements in the SNA environment are required in order for two TPs to communicate with each other. The following figure illustrates several fundamental elements:



Each TP is associated with a logical unit (LU) of type 6.2. The LU allows the TP to access the network. Note that several TPs can be associated with the same LU.

A partner TP can invoke another TP, which, in turn, invokes another TP, and so on. In the following figure, TP A invokes TP B, and TP B invokes TP C.



This section contains:

- Fundamental Terms for TPs and LUs
- Sample TPs Illustrating Fundamental Concepts
- Configuring and Controlling TPs
- Creating TPs and Their Supporting Configuration

Fundamental Terms for TPs and LUs

The following terms describe some fundamental characteristics of TPs communicating through LUs:

basic conversation
A type of conversation more complex than a mapped conversation and generally used by service TPs (SNA-based programs that provide services to other programs). For a basic conversation, use Set_Conversation_Type and specify CM_BASIC_CONVERSATION for the *conversation_type*. For more information, see Basic and Mapped Conversations Compared.

conversation
The interaction between TPs carrying out a specific task. Each conversation requires an LU-LU session. A TP can be involved in several conversations simultaneously, as shown with TP B in Communication Between TPs.

invokable TP
A TP that can be invoked by another TP. Invokable TPs are usually server-type applications, that is, they work in the same general way that an IBM CICS application works. Parameters for an invokable TP are configured through registry or environment variables.

There are several types of invokable TPs:

operator-started invokable TP

A TP that is started manually in preparation for being invoked.

autostarted invokable TP

A TP that is automatically started by CPI-C when invoked.

queued TP

A TP that, when invoked multiple times, loads once and then queues up subsequent requests to be dealt with one at a time. All operator-started TPs and some autostarted TPs are queued.

nonqueued TP

A TP loaded multiple times, once for every time it is invoked. Some autostarted TPs are nonqueued but no operator-started TPs are nonqueued.

For more information see Invokable TPs.

invoking TP
A TP that can invoke (that is, initiate a conversation with) other TPs. Invoking TPs are usually client-type applications, that is, they work in the same general way that an emulator works. For more information see Invoking TPs.

local LU and **local TP**
An LU and TP working together, when viewed as the "home base" for a particular conversation. From this viewpoint, some other LU and TP are seen as the "partner" or "remote" LU and TP.

LU alias
The string that identifies an LU to a TP. The alias can be up to eight characters long.

LU-LU session
The communication between two LUs over a specific connection for a specific amount of time. An LU-LU session is needed for two TPs to interact. One session can be used serially by many pairs of TPs.

An LU 6.2 can have multiple sessions (two or more concurrent sessions with different partner LUs) and parallel sessions (two or more concurrent sessions with the same partner LU).

LUs are configured through SNA Management on Host Integration Server or through SNA Server Manager on the earlier SNA Server. These administration tools are also used to configure LU-LU pairs and modes; the LU and mode configurations control how many sessions a particular LU-LU pair supports.

mapped conversation
A type of conversation simpler than a basic conversation and generally used by application TPs (programs that accomplish tasks for end users). The default for conversation type is mapped; the conversation type can be changed with the Set_Conversation_Type call. For more information, see Basic and Mapped Conversations Compared.

partner LU and **partner TP**, or **remote LU** and **remote TP**
An LU and TP working together, when viewed as being at the far end of a particular conversation.

Sample TPs Illustrating Fundamental Concepts

A set of sample TPs is provided on the Host Integration Server CD-ROM in the \SDK\Samples\SNA directory (in the \SDK\SAMPLES directory on the earlier SNA Server CD-ROM). Included with the sample code is TPSETUP, a program that simplifies the setting of registry or environment variables needed by autostarted invokable TPs. Without an interface like that provided by TPSETUP, configuring such variables can be complicated and error-prone. Therefore, it is recommended that you use code like TPSETUP in installation programs for autostarted invokable TPs.

INSTALL.C (the source code for TPSETUP) can be compiled to work either in the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 environments or in the Windows version 3.*x* environment.

For information about TPSETUP and about the sample TPs, see Sample CPI-C TPs in the SDK.

Configuring and Controlling TPs

The following table shows how the characteristics of the TPs and selection of the LUs for a conversation are controlled.

Characteristic	How controlled
Type of conversation: basic or mapped	Written into the code. For two TPs to communicate successfully, both must use the same type of conversation, basic or mapped. The default for conversation type is mapped; the type can be changed with the Set_Conversation_Type call. See Basic and Mapped Conversations Compared.
Type of TP: invoking or invokable	Written into the code. Invoking TPs start with Initialize_Conversation and Allocate. Invokable TPs start with Accept_Conversation. See Invoking TPs and Invokable TPs.
The local LU alias to be used by an invoking TP	Three options: • Configured with a registry or environment variable. • Configured (in Host Integration Server using SNA Management or in SNA Server using SNA Server Manager) as the default local APPC LU for the user who starts the invoking TP. • Configured (in Host Integration Server using SNA Management or in SNA Server using SNA Server Manager) as a member of the default outgoing local APPC LU pool. See Invoking TPs and the SNA Server Configuration.
The symbolic destination name used by an invoking TP	Written into the code, in Initialize_Conversation.
The invokable (partner) TP requested by an invoking TP	Specified within the symbolic destination name, which can be configured through SNA Management using Host Integration Server or SNA Server Manager on the earlier SNA Server product.
The LU alias to be used by an invokable TP (the partner LU alias from the point of view of the invoking TP)	Specified within the symbolic destination name, which can be configured through SNA Management using Host Integration Server or SNA Server Manager on the earlier SNA Server product. See Invoking TPs and the SNA Server Configuration and Matching Invoking and Invokable TPs.
Type of autostarted invokable TP: queued or nonqueued	Configured with registry or environment variables. See Configuring Invokable TPs.
Local LU and remote LU aliases	Configured through SNA Management using Host Integration Server or SNA Server Manager on the earlier SNA Server product. For information, see the Installation and Configuration section in the *Microsoft Host Integration Server Guide* or the earlier *Microsoft SNA Server Administration Guide*.
The pairing of local and remote LUs, and the mode used for each LU-LU pair	Configured through SNA Management using Host Integration Server or SNA Server Manager on the earlier SNA Server product. For information, see the Installation and Configuration section in the *Microsoft Host Integration Server Guide* or the *Microsoft SNA Server Administration Guide*.

Creating TPs and Their Supporting Configuration

The following procedure describes how to create TPs and set up a supporting configuration.

To create TPs and set up a supporting configuration

1. Write, compile, and link each TP.
2. Place each TP on an appropriate computer.

   For TPs that you will start many times or that will be started by a user, arrange for the TP to be started easily. That is, for graphical interfaces, create a program icon for starting the TP; for nongraphical interfaces, make sure the TP is in the path.

3. On one or more SNA servers, configure LUs, modes, LU-LU pairs, and a symbolic destination name for use by the TPs.

   For information about how to set up LU-LU pairs to support TPs, see Using Invoking and Invokable TPs and the Host Integration Server or SNA Server. For information, see the Installation and Configuration section in the *Microsoft Host Integration Server Guide* or the *Microsoft SNA Server Administration Guide*.

   For information about symbolic destination names and side information, see Side Information and the Host Integration Server or SNA Server. For information, see the Installation and Configuration section in the *Microsoft Host Integration Server Guide* or the *Microsoft SNA Server Administration Guide*.

4. Set any registry or environment variables needed for the invoking and invokable TPs.

   For autostarted invokable TPs, it is recommended that you use the sample TP configuration program, TPSETUP, for this step. When you write an installation program for autostarted invokable TPs, it is recommended that you include code similar to TPSETUP.

   For information about registry or environment variables, see Configuring Invokable TPs and Invoking TPs. For information about TPSETUP, see Sample CPI-C TPs in the SDK.

5. If the invokable TP is operator-started, start it, or arrange for it to be started when the computer is restarted and then restart the computer.

   If the invokable TP is autostarted, Host Integration Server or SNA Server will start it when needed.

6. Start the invoking TP.

Designing and Coding TPs

The following topics provide background information about designing and coding TPs.

This section contains:

CPI-C Calls in C Programs

This implementation of CPI-C is available to programs written in Microsoft® C version 5.1 or later.

The WINCPIC.H header file defines the prototypes for each CPI-C function. Other definitions include:

- Types specifically defined for use by CPI-C parameters.
- The structure of the side information entries.
- Symbolic names defined for integer parameters.

To use CPI-C calls, the C program must include WINCPIC.H and declare the variables to be used in passing parameters on CPI-C calls. Note that you must define WIN32, WINDOWS, or DOS5 before including WINCPIC.H. For example:

```
Windows 2000, Windows NT, Windows 98, and Windows 95 clients:
        #define WIN32
        #include <wincpic.h>

Windows 3.x clients:
        #define WINDOWS
        #include <wincpic.h>

OS/2 clients:
        #define DOS5
        #include <wincpic.h>
```

In the case of strings, the program must also determine the desired string length.

CPI-C and LU 6.2

CPI-C applications can communicate with non-CPI-C LU 6.2 applications, such as APPC.

CPI-C supports all functions of LU 6.2 except:

- Sync Point/backout processing
- PIP data
- LOCKS=LONG
- MAP_NAME
- FMH_DATA

Conversation States

The state of the conversation (as viewed by a particular TP) governs which CPI-C calls can be made by the TP at a particular time. For example, a TP cannot issue Send_Data if the conversation is not in SEND or SEND_PENDING state for that TP.

The state of a conversation depends on the TP from which it is viewed. A local TP can view a conversation as being in SEND state while the partner TP views the conversation as being in RECEIVE state. A particular TP can be in several conversations, each of which is in a different state.

The possible conversation states are summarized here.

CONFIRM
The TP has received a request for confirmation of receipt of data; it must respond positively or send error information to the partner TP.

CONFIRM_DEALLOCATE
The TP has received a request for confirmation and must respond positively or send error information. If the TP responds positively, the conversation is automatically deallocated.

CONFIRM_SEND
The TP has received a request for confirmation; it must respond positively or send error information. After responding, the TP can begin to send data.

INITIALIZE
The conversation has been initialized successfully.

RECEIVE
The TP can receive application data and status information from the partner TP. When the conversation is in RECEIVE state, the TP can also send error information and request permission to send data.

RESET
The conversation has not started or has been terminated.

SEND
The TP can send data to the partner TP and request confirmation. When the conversation is in SEND state, the TP can also begin to receive data, which can cause the state to change to RECEIVE.

SEND_PENDING
The TP issued a Receive call and received data as well as a send indicator (*status_received* = CM_SEND_RECEIVED), indicating that the TP can begin to send data. This state differs from the SEND state, which occurs when the TP receives data on one **Receive** call and the send indicator on a subsequent **Receive** call.

This section contains:

- State Checks
- Changing Conversation States

State Checks

A state check occurs when a TP issues a CPI-C call and the conversation is not in the appropriate state. For example, a state check occurs if a TP issues Send_Data while the conversation is in RECEIVE state. When a state check occurs, CPI-C does not execute the call; it returns state check information through the *return_code* parameter.

Changing Conversation States

A change in the conversation state can result from:

- A call made by the local TP.
- A call made by the partner TP.
- An error condition.

The following example shows how CPI-C calls can change the state of the conversation from SEND to RECEIVE and from RECEIVE to SEND.

> Any TP can send or receive data, regardless of whether it is the invoking TP (the TP that started the conversation) or the invokable TP (the TP that responded to a request to start a conversation).

This example shows how CPI-C calls can change the conversation state. In this table, each conversation state appears in bold and precedes the CPI-C calls that are used while in that state.

Issued by the invoking TP	**Issued by the invokable TP**
Conversation state: RESET	
Initialize_Conversation	
Conversation state: INITIALIZE	
Set_Sync_Level	
(sync_level=CM_CONFIRM)	
Allocate	
Conversation state: SEND	
Send_Data	
Prepare_to_Receive	**Conversation state: RESET**
	Accept_Conversation
	Conversation state: RECEIVE
	(status_received= CM_CONFIRM_SEND_RECEIVED)
	Conversation state: CONFIRM_SEND
	Confirm
	Conversation state: SEND
(return_code=CM_OK)	Send_Data
Conversation state: RECEIVE	Confirm
(status_received= CM_CONFIRM_RECEIVED)	
Conversation state: CONFIRM	
Request_To_Send	
Confirmed	
Conversation state: RECEIVE	(return_code=CM_OK)
	(request_to_send_received= CM_REQ_TO_SEND_RECEIVED)
	Prepare_To_Receive
Receive	
(status_received= CM_CONFIRM_SEND_RECEIVED)	
Conversation state: CONFIRM_SEND	
Confirmed	
Conversation state: SEND	(return_code=CM_OK)
	Conversation state: RECEIVE
Send_Data	
Deallocate	
	Receive
	(status_received= CM_CONFIRM_DEALLOC_RECEIVED)

	Conversation state: CONFIRM_DEALLOCATE
	Confirmed
(return_code=CM_OK)	**Conversation state: RESET**
Conversation state: RESET	

Initial States

Before the conversation is allocated, the state is RESET for both TPs.

In the example, after the conversation is allocated, the initial state is SEND for the invoking TP and RECEIVE for the invokable TP.

Changing to RECEIVE State

The Prepare_To_Receive call allows a TP to change the conversation from SEND to RECEIVE state. This call:

- Flushes the local LU's send buffer.
- Sends a CM_CONFIRM_SEND indicator to the partner TP through the *status_received* parameter of a Receive call, because the synchronization level is set to CM_CONFIRM. This indicator tells the partner TP that a Confirmed response is expected before the partner TP can begin to send data.

Changing to SEND State

The Request_To_Send call informs the partner TP (for which the conversation is in SEND state) that the local TP (for which the conversation is in RECEIVE state) wants to send data. This request is communicated to the partner TP through the *request_to_send_received* parameter of the Confirm call. (The *request_to_send_received* parameter is also returned to Send_Data and other calls.)

When the partner TP issues the Prepare_To_Receive call, the conversation state changes to RECEIVE for the partner TP, making it possible for the local TP to send data.

> **Important** Issuing Request_To_Send does not change the state of the conversation. Upon receiving a request to send, the partner TP is not required to change the conversation state; it can ignore the request.

Confirmation Processing

The sequence of events for confirmation processing is as follows:

1. Establish the synchronization level.
2. Send a confirmation request.
3. Receive data and confirmation request.
4. Respond to the confirmation request.
5. Deallocate the conversation.

Using confirmation processing, a TP sends a confirmation request with the data; the partner TP confirms receipt of the data or indicates that an error occurred. Each time the two TPs exchange a confirmation request and response, they are synchronized.

> Although the example in this section does not show this, any TP can send or receive data, regardless of whether the TP is the invoking TP or the invokable TP.

The following example illustrates confirmation processing.

Issued by the invoking TP	**Issued by the invokable TP**
Initialize_Conversation	
Set_Sync_Level (*sync_level*=CM_CONFIRM)	
Allocate	
Send_Data	
Confirm	
	Accept_Conversation
	Receive (data_received= CM_COMPLETE_DATA_RECEIVED) (status_received= CM_CONFIRM_RECEIVED)
	Confirmed
(*return_code*=CM_OK)	
Send_Data	
Deallocate	
	Receive
	(status_received= CM_CONFIRM_DEALLOC_RECEIVED)
	Confirmed
(return_code=CM_OK)	

Establishing the Synchronization Level

The Set_Sync_Level call lets you override the default synchronization level of the conversation. The synchronization level is one of the conversation's characteristics. There are two possible synchronization levels:

- CM_CONFIRM, under which the TPs can request confirmation of receipt of data and respond to such requests.
- CM_NONE, the default, under which confirmation processing does not occur.

The Initialize_Conversation call sets the default characteristics of a conversation. There are several calls that begin with **Set_**. These calls let you override the default conversation characteristics.

Sending a Confirmation Request

Issuing the Confirm call has two effects:

- It flushes the local LU's send buffer and sends any data contained in the buffer to the partner TP.
- It sends a confirmation request that the partner TP receives through the *status_received* parameter of a Receive call.

After issuing **Confirm**, the local TP waits for confirmation from the partner TP.

Receiving a Confirmation Request

The *status_received* parameter of the Receive call indicates any future action required by the local TP.

In the example, the first Receive has a *status_received* of CM_CONFIRM_RECEIVED, indicating that a confirmation is required before the partner TP can continue.

Responding to a Confirmation Request

The partner TP issues the Confirmed call to confirm receipt of data. This frees the local TP to resume processing.

Deallocating the Conversation

Because the synchronization level of the conversation is set to CM_CONFIRM, Deallocate sends a confirmation request with the data flushed from the buffer.

For the second Receive call, *status_received* is CM_CONFIRM_DEALLOC_RECEIVED, indicating that the partner TP requires a confirmation, generated by the Confirmed call, before the conversation can be deallocated.

Conversation Security

You can use conversation security to require that the invoking TP provide a user identifier and password before CPI-C will allocate a conversation with the invokable TP.

For the invoking TP, conversation security is activated and configured (with user identifier and password) through the symbolic destination name in Microsoft® SNA Server Manager or by the following calls, which override the symbolic destination name:

- Set_Conversation_Security_Type
- Set_Conversation_Security_User_ID
- Set_Conversation_Security_Password

For the invokable TP, conversation security is activated and configured through registry or environment variables on the computer where the invokable TP is located.

With communication involving more than two TPs, the verification of a user identifier and password can be passed from one TP to another. Suppose that TP A invokes TP B, which requires security information, and TP B in turn invokes TP C, which also requires security information. TP B can inform TP C that conversation security has already been verified.

For information about the registry or environment variables affecting conversation security, see Configuring Invokable TPs. For information about symbolic destination names and side information, see Side Information and the *Microsoft SNA Server Administration Guide*.

Basic and Mapped Conversations Compared

The following table offers some guidelines for choosing between basic and mapped conversations for your TPs. The default for conversation type is mapped; to change to a basic conversation, use Set_Conversation_Type, and specify CM_BASIC_CONVERSATION for the *conversation_type*. For definitions of basic and mapped conversations, see Fundamental Terms for TPs and LUs.

Characteristic	Basic conversations	Mapped conversations
Common use	Generally used for service TPs.	Generally used for application TPs.
Partnering	Must be used to communicate with an existing TP that uses basic verbs.	Must be used to communicate with an existing TP that uses mapped verbs.
Sending and receiving method	Before a TP can begin a send operation, it must convert data records into logical records. The TP does this by adding a 2-byte prefix that indicates the length of the record. A TP can send several logical records at one time. When a partner TP receives logical records, it must reconstruct them into usable data records. For more information, see Logical Records Used in Basic Conversations.	A TP sends data one record at a time. Neither the sending TP nor the receiving TP needs to convert data records between different forms.
Abnormal termination	In the Deallocate call, a TP can indicate whether an error or ABEND (abnormal program termination) was caused by a TP or by a program using the TP.	A TP can indicate an error or ABEND, but cannot tell whether a problem was caused by a TP or by a program using a TP.
	A TP can indicate whether an ABEND was caused by a timeout or by a critical error.	A TP cannot indicate the cause of an ABEND.
Error logging	For an error or ABEND, a TP can send an error message, in the form of a general data stream (GDS) error log variable, to the local log and to the partner LU.	For an error or ABEND, a TP cannot send an error message to the local log or to the partner LU.

This section contains:

- Logical Records Used in Basic Conversations
- An Example of a Mapped Conversation

Logical Records Used in Basic Conversations

Logical records are sent and received in basic conversations only.

A TP can send or receive multiple logical records with a single Send_Data or Receive call. A TP can also send or receive a logical record in successive portions: beginning, middle, and end.

A logical record is made up of:

- A 2-byte record-length (LL) field.
- A data field that can range in length from 0 bytes through 32765 bytes.

The LL field contains a hexadecimal value that is the length of the data field plus two bytes (for the LL field). For example, if a record contains 228 bytes of application data, the logical record length is 230. The LL field is 0x00E6, the hexadecimal equivalent of 230. If the length of the data field is 0, the value contained in the LL field is 0x0002.

Logical records are sent from or received in a data buffer established by the TP. In the data buffer, the LL field must not be in Intel byte-swapped format. For example, a length of 230 must be 0x00E6, not 0xE600.

The LL field cannot be 0x0000 or 0x0001, which allow less than the two bytes required for the LL field itself. The LL field also cannot be greater than or equal to 0x8000, which is equivalent to decimal 32768 and therefore allows for a data field greater than 32765 or an LL field greater than 2.

Setting the most significant bit of the LL field to 1 indicates that the information contained in the current logical record is continued in the next logical record.

An Example of a Mapped Conversation

The following example of a mapped conversation shows the CPI-C calls used to start a conversation, exchange data, and end the conversation. Call parameters are in parentheses.

Issued by the invoking TP	**Issued by the invokable TP**
Initialize_Conversation	
Allocate	
Send_Data	
Deallocate	Accept_Conversation
	Receive
	(data_received=
	CM_COMPLETE_DATA_RECEIVED)
	(*return_code*=
	CM_DEALLOCATED_NORMAL)

The following paragraphs describe the calls that are used in a mapped conversation.

Calls for Starting a Mapped Conversation

To start a conversation, the invoking TP issues the following calls:

- Initialize_Conversation, which requests CPI-C to set the values defining the characteristics of the conversation. The **Initialize_Conversation** call specifies a symbolic destination name that is associated with an entry in a side information table in memory. The side information specifies partner TP, partner LU, mode, security, and so on.
- Allocate, which requests that CPI-C establish a conversation between the invoking TP and the invokable TP.

The invokable TP issues the Accept_Conversation call, which informs CPI-C that it is ready to begin a conversation with the invoking TP.

Calls for Sending Data in a Mapped Conversation

The Send_Data call puts one data record (a record containing application data to be transmitted) in the send buffer of the local LU. Data transmission to the partner TP does not happen until one of the following events occurs:

- The send buffer fills up.
- The sending TP makes a call that forces CPI-C to flush the buffer and send data to the partner TP.

In addition to the data record, the send buffer also contains the allocation request (which precedes the data record).

In the preceding example, Deallocate flushes the send buffer, sending the allocation request and data to the partner TP. Other calls that flush the buffer are Confirm and Flush.

Calls for Receiving Data in a Mapped Conversation

The **Receive** call receives the data record and status information from the partner TP. If no data or status information is currently available, the local TP, by default, waits for data to arrive.

The *data_received* parameter of Receive tells the program whether it received data and if so, whether or not the data is complete.

Calls for Ending a Mapped Conversation

To end a conversation, one of the TPs issues Deallocate, which causes CPI-C to deallocate the conversation between the two TPs.

Using Invoking and Invokable TPs

There are two kinds of TPs: TPs that can invoke (that is, initiate a conversation with) other TPs, and TPs that can be invoked. A TP that can invoke another TP is called an invoking TP, and a TP that can be invoked is called an invokable TP.

The following topics describe how:

- Invoking TPs request invokable TPs.
- Invokable TPs identify themselves to SNA Server in preparation for being invoked.
- An invokable TP is matched to an invoking TP's request.

For information about how to configure LUs to support TPs, see Configuring Host Integration Server to Support TPs and the Installation and Configuration section of the *Microsoft Host Integration Server Guide Microsoft SNA Server Administration Guide* (for SNA Server, see the *Microsoft SNA Server Administration Guide*).

This section contains:

Invoking TPs

An invoking TP can be located on any system on the SNA network. An invoking TP identifies itself by issuing Initialize_Conversation, which specifies the name of the invoking TP and the symbolic destination name to be used. A local LU alias can be specified for the invoking TP by using a registry or environment variable, as shown in the following table.

Operating system on computer that contains invoking TP	**Location and name of variable**
Microsoft® Windows 2000 and Windows NT®	Location in Windows 2000 or Windows NT registry: HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters Client <*exename*>:REG_SZ:*localLUalias* Any *exename* registry entries under the Client key represent the file names of Win32 executable files (without the file extension) for any Invoking TPs. A REG_SZ value associated with each *exename* registry entry specifies the local LU alias for the invoking TP. For example, the APING.EXE CPI-C sample included with the Host Integration Server or earlier SNA Server SDK would have the following registry entry: HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters Client APING:REG_SZ:*localLUalias*
Microsoft® Windows® 98 and Windows® 95	Location in Windows 98 or Windows 95 registry: HKEY_LOCAL_MACHINE SOFTWARE Microsoft SnaBase Parameters Client <*exename*>:REG_SZ:*localLUalias* Any *exename* registry entries under the Client key represent the file names of Win32 executable files (without the file extension) for any Invoking TPs. A REG_SZ value associated with each *exename* registry entry specifies the local LU alias for the invoking TP. For example, the APING.EXE CPI-C sample included with the Host Integration Server or earlier SNA Server SDK would have the following registry entry: HKEY_LOCAL_MACHINE SOFTWARE Microsoft SnaBase Parameters Client APING:REG_SZ:*localLUalias*
Windows version 3.*x*	Section and variable in WIN.INI file: [*ApplicationName*] APPCLLU=*localLUalias*

OS/2	Section and variable in SNA.INI file: [*ApplicationName*] APPCLLU=*localLUalias* Alternatively, this can be configured through set commands, either at the command prompt or in CONFIG.SYS.

The registry parameter for the local LU alias takes greatest precedence when associating a local LU to an invoking CPI-C application. If a registry value is not configured, two other methods are used to associate a local LU to the CPI-C application:

A local APPC LU can be associated with the user context under which the CPI-C application is running, A local APPC LU can be configured by checking the "member of default local APPC LU pool" checkbox. Of the two possible options, a local LU associated with user context has the higher precedence.

If the local LU alias is not specified in a registry or environment variable, SNA Server must be configured to supply it through one of these two types of default local LU; otherwise, Initialize_Conversation will fail. For more information, see Invoking TPs and the SNA Server Configuration.

Next, the symbolic destination name specified in Initialize_Conversation provides the name of the invokable (or partner) TP and the partner LU alias (the LU alias to be used by the invokable TP). With this information available, the invoking TP can issue the Allocate call.

After a TP successfully issues an Allocate call, an allocation request flows. For more information about what happens after an invoking TP requests an invokable TP, see Matching Invoking and Invokable TPs.

Invoking TPs and Contention

The following information applies only to cases where LUs are communicating in complex ways (such as chains of LUs) over multiple sessions. In such cases, two LUs may attempt to allocate a conversation on the same session at the same time. If this happens, one LU must win (the contention winner) and one must lose (the contention loser). The contention-winner LU and the contention-loser LU are determined for each session when the session is established. During that particular session, the contention-loser LU must receive permission from the contention-winner LU before allocating a conversation. In contrast, the contention-winner LU on that session allocates a conversation as needed.

Note that when two LUs are communicating over multiple sessions, one LU can be the contention winner for some of the sessions, and the other LU the contention winner for others.

An invoking TP will operate most efficiently if the number of concurrent Allocate requests that the TP issues is matched by the number of sessions on which the local LU is the contention winner. The choice of contention winner is controlled through the modes configured at the two ends of the communication. For SNA Server, the mode is configured in SNA Server Manager. A mode must be configured to work with the mode on the remote system for communication to begin between two LUs. For more information about modes, see the *Microsoft SNA Server Administration Guide*.

Invokable TPs

An invokable TP is a TP that can be invoked by another TP. Invokable TPs are written or configured through registry or environment variables to supply their names to SNA Server as a notification that they are available for incoming requests. SNA Server invokable TPs can be run on any SNA server or client running Windows NT, Windows 95, Windows version 3.*x*, or OS/2.

There are two types of invokable TPs:

Operator-started invokable TPs

An operator-started invokable TP must be started by an operator before the TP can be invoked. When the operator-started invokable TP is started, it notifies SNA Server of its availability by issuing an Accept_Conversation call. The **Accept_Conversation** call causes the name of the invokable TP to be communicated to all the SNA servers in the domain, along with the alias of an associated LU if one has been configured through a registry or environment variable.

Autostarted invokable TPs

An autostarted invokable TP can be started by SNA Server when needed. The TP must be registered through registry entries or environment variables on its local system, so that it can be identified to the SnaBase component of the SNA Server client software. The registered information defines the TP as autostarted and must specify the TP name. The registered information can also specify the local LU alias that the invokable TP will use.

The recommended method for setting registry or environment variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP, or similar code written into your own installation program. For more information about registry or environment variables for invokable TPs, see Configuring Invokable TPs. For information about TPSETUP, see Sample CPI-C TPs in the SDK.

If no local LU alias is registered with autostarted TPs, the resulting SNA Server configuration can be more flexible in responding to invoking requests. For more information about such flexible configurations, see TP Name Not Unique; Local LU Alias Unspecified.

After an autostarted invokable TP is started by SNA Server, the TP issues Accept_Conversation just as an operator-started TP does. Accept_Conversation must provide the TP name that was registered for the TP.

Autostarted TPs must be configured through registry or environment variables to be either queued or nonqueued. All operator-started TPs act as queued TPs.

Queued TPs

If an autostarted TP is configured as queued, or if the TP is operator-started, incoming allocation requests are queued and then sent only when the invokable TP issues **Accept_Conversation**. For autostarted invokable TPs, if a copy of the TP is not yet running, one is started when an incoming allocation request specifies that TP.

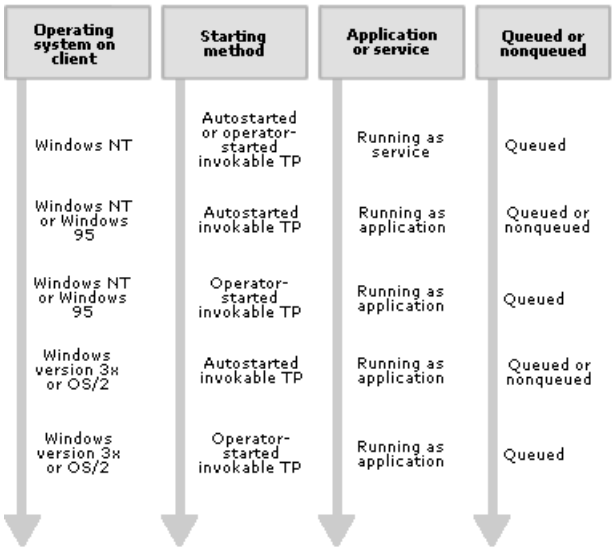
> For the Windows 2000 and Windows NT system, only one copy of a service can be running at any given time; this means that all autostarted TPs that run as services under Windows 2000 or Windows NT must be queued. To write an autostarted TP so it will run under Windows 2000 and Windows NT as a service and also run in a nonqueued way, write a multithreaded program with an **Accept_Conversation** always outstanding.

Nonqueued TPs

If an autostarted TP is configured as nonqueued, a new copy will be started every time an Allocate is received for the TP. Nonqueued TPs should process the conversation they have been allocated and then exit, since they will not receive any additional **Allocate** requests.

Subcategories for Invokable TPs

The following figure shows subcategories for invokable TPs.



The concept of a TP "running as a service" or "running as an application" is distinct from a service TP or an application TP. Service TP and application TP are SNA terms that describe how a TP is used: either as a supportive service program for other CPI-C programs, or directly by a user, as an application. For detailed information about services in Windows 2000 and Windows NT, see the documentation for Windows 2000 and Windows NT and the Microsoft® Developer Network (MSDN) Platform Software Development Kit (SDK).

To write an autostarted TP so it will run under Windows NT as a service and also run in a nonqueued way, write a multithreaded program with an Accept_Conversation always outstanding. See Invokable TPs.

To run an autostarted TP as an application under Windows 2000, Windows NT, Windows 98 or Windows 95, make sure the TPSTART program is always started before the TP. See Sample CPI-C TPs in the SDK.

Matching Invoking and Invokable TPs

Each Host Integration Server or SNA Server maintains a list of available invokable TP names and any LU aliases to be associated with the TP names. This information is obtained as follows:

- For autostarted invokable TPs, registry or environment variables identify a TP name containing a maximum of eight characters, and can specify an associated LU. This information is sent from the client to the server that sponsors the client. A client learns about the domain through a sponsor connection to a server; clients must establish the sponsor connection before proceeding with any other tasks.
- For operator-started invokable TPs, a TP name (with a maximum of 64 characters) is specified in Specify_Local_TP_Name (or, for OS/2 only, by setting APPCTPN=*TPname*). The TP name is truncated to eight characters and sent from the client to the server that sponsors the client, along with the alias of an associated LU if one has been configured through a registry or environment variable.

> **Note** If you want a TP name to be unique, it is recommended that you limit the name to eight characters or fewer, or make the name unique within the first eight characters. This is because the preliminary routing of allocation requests is carried out using the first eight characters. Although further matching is later carried out between the full TP names, it is inefficient to allow the preliminary routing to succeed when in some cases the later matching will fail.

The next step in the matching of invoking and invokable TPs is the creation of a side information table from the parameters in the symbolic destination name. Then the invoking TP issues the Allocate call, and an allocation request flows to the partner LU specified in the side information table, stating the name of the invokable TP that has been requested (also listed in the side information table).

When an allocation request arrives, the SNA server compares the requested invokable TP name and LU alias to the list of available invokable TPs (which can include associated LU aliases). The comparison can be modified by registry variables, but by default is carried out as follows:

- Although the TP name requested in the symbolic destination name can be as long as 64 characters, any name received through a registry or environment variable is limited to eight characters or less. Therefore, only the first eight characters of TP names are used in comparisons.
- The comparison is carried out first on both the TP name and the LU alias. An invokable TP for which there is a match on both TP name and LU alias will be chosen ahead of a TP for which no LU alias has been configured through a registry or environment variable. A TP for which no LU alias has been configured can be matched with any request that specifies that TP name, since there cannot be a mismatch based on LU alias.
- The comparison of requested and available TP names is carried out in a specific order:

  1. The Host Integration Server or SNA Server first checks for operator-started invokable TPs on the local system (the local SNA server).
  2. If no match is found, the Host Integration Server or SNA Server checks for autostarted invokable TPs on the local system (the local SNA server).
  3. If no match is found, the Host Integration Server or SNA Server checks for operator-started invokable TPs on other SNA servers or clients.
  4. If no match is found, the Host Integration Server or SNA Server checks for autostarted invokable TPs on other SNA servers or clients.

This comparison can be modified somewhat by registry entries for the SnaServr service. The entries are called **DloadMatchTPOnly** and **DloadMatchLocalFirst**, and are described in the Installation and Configuration section of the *Microsoft Host Integration Server Guide* or the earlier *Microsoft SNA Server Reference*.

If a match is found, the SNA server signals the system containing the requested TP to connect to that SNA server. If no match is found, the SNA server rejects the incoming request.

For suggestions about specific ways to handle TP names and LU aliases, see Arranging TPs Within an SNA Network.

> **Note** Because of the way CPI-C works, an allocation request will not flow until local data buffers are full, or a Confirm or Flush call is made. This may mean that the allocation request does not flow until some time after the Allocate call is made. Therefore, any allocation failure caused by the rejection of the allocation request at the partner LU will be observed as the failure of a later call with one of the allocation failure return codes.

Configuring Invokable TPs

The following topics tell how to configure invokable TPs for the various Microsoft® Host Integration Server and SNA Server client types.

This section contains:

Clients Running Windows 2000 or Windows NT

On clients running Microsoft® Windows 2000 or Microsoft® Windows NT®, invokable TPs are configured through the Windows NT registry.

> **Note** With Windows 2000 or Windows NT, the recommended method for setting registry variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile INSTALL.C, the source code for TPSETUP, for the Windows 2000 and Windows NT environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample CPI-C TPs in the SDK.

For clients running Windows 2000 or Windows NT, it is recommended that autostarted invokable TPs be written as Windows NT services. Be sure to include code like that in TPSETUP in the program that installs your TPs. Among other things, TPSETUP shows how to use the **CreateService** function when installing a TP. For important information about how services work under Windows 2000 and Windows NT, see the documentation for Windows 2000 and Windows NT and Microsoft Platform SDK.

The following table lists the registry entries used for the types of invokable TPs that can be run on Windows 2000 and Windows NT clients:

Type of TP	**Location in registry**	**Possible registry entries**
Autostarted invokable TP running as a service on Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services *TPName*** (and subkeys)	Registry entries created by the **CreateService** call, including entries that specify the path, display name, and other characteristics of the service. —plus— Linkage OtherDependencies:REG_MULTI_SZ:SnaBase Parameters SNAServiceType:REG_DWORD:0x5 LocalLU:REG_SZ:*LUalias* Parameters:REG_SZ:*ParameterList* Timeout:REG_DWORD:*number* AcceptNames:REG_SZ:*TPNameList* ConversationSecurity:REG_SZ:{ YES | NO } AlreadyVerified:REG_SZ:{ YES | NO }(2) *Username1*:REG_SZ:*Password1*(2) ... *UsernameX*:REG_SZ:*PasswordX*(2)
Autostarted invokable TP running as an application1 on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters TPs *TPName*** Parameters	**SNAServiceType:**REG_DWORD:{ 0x5 | 0x6 } **PathName:**REG_EXPAND_SZ:*path* **LocalLU:**REG_SZ:*LUalias* **Parameters:**REG_SZ:*ParameterList* **TimeOut:**REG_DWORD:*number* **AcceptNames:**REG_SZ:*TPNameList* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }(2) ***Username1*:**REG_SZ:*Password1*(2) ... ***UsernameX*:**REG_SZ:*PasswordX*(2)

Operator-started invokable TP running as a service on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services *TPName*** (and subkeys)	Registry entries created by the **CreateService** call, including entries that specify the path, display name, and other characteristics of the service. —plus— Linkage OtherDependencies:REG_MULTI_SZ:SnaBase Parameters SNAServiceType:REG_DWORD:0x1A LocalLU:REG_SZ:*LUalias* Timeout:REG_DWORD:*number* ConversationSecurity:REG_SZ:{ YES | NO } AlreadyVerified:REG_SZ:{ YES | NO }(2) *Username1*:REG_SZ:*Password1*(2) ... *UsernameX*:REG_SZ:*PasswordX*(2)
Operator-started invokable TP running as an application on a Windows 2000 or Windows NT client	**HKEY_LOCAL_MACHINE SYSTEM CurrentControlSet Services SnaBase Parameters TPs *TPName*** Parameters	**SNAServiceType:**REG_DWORD:0x1A **LocalLU:**REG_SZ:*LUalias* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }(2) ***Username1*:**REG_SZ:*Password1*2 ... ***UsernameX*:**REG_SZ:*PasswordX*(2)

Notes

1. Before an autostarted TP can be started as an application on a Windows NT-based client, the TPSTART program must be started. For more information, see Sample CPI-C TPs in the SDK.

2. **AlreadyVerified** and Username/Password entries are used only if **ConversationSecurity** is set to YES.

This section contains:

- Registry Entries for Clients Running Windows 2000 or Windows NT
- Example of Registry Entries for Windows 2000 or Windows NT

Registry Entries for Clients Running Windows 2000 or Windows NT

The following list gives details about registry entries for clients running Windows 2000 or Windows NT. For each TP type, the applicable variables and their locations are shown in Clients Running Windows 2000 or Windows NT.

OtherDependencies:REG_MULTI_SZ:SnaBase
For a TP running as a service, ensures that the SnaBase service will be started before the TP is started. This entry belongs under the **Linkage** subkey.

SNAServiceType:REG_DWORD:{ 0x5 | 0x6 | 0x1A }
Indicates the type of TP. Use a value of 0x5 for an autostarted queued TP, 0x6 for an autostarted nonqueued TP, and 0x1A for an operator-started TP.

Note that the value for an autostarted TP running as a service must be 0x5, because these TPs are always queued, as described in Invokable TPs.

PathName:REG_EXPAND_SZ:*path*
For an autostarted TP running as an application, specifies the path and file name of the TP. The data type of REG_EXPAND_SZ means that the path can contain an expandable data string; for example, %SystemRoot% represents the directory containing the Windows 2000 or Windows NT system files. Note that for a TP running as a service, an equivalent entry is inserted by the **CreateService** call; no additional path entry is needed.

LocalLU:REG_SZ:*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters:REG_SZ:*ParameterList*
Lists parameters to be used by the TP. Separate parameters with spaces.

Timeout:REG_DWORD:*number*
Specifies the time, in milliseconds, that an Accept_Conversation will wait before timing out. Specify *number* in decimal; the registry editor converts this to hexadecimal before displaying it. The default is infinity (no limit).

AcceptNames:REG_SZ:*TPNameList*
With Windows NT, used for autostarted TPs only; lists additional names under which the invokable TP can be invoked. Separate TP names with spaces. The default is none. If an invokable TP does not issue a Specify_Local_TP_Name for each name configured under AcceptNames in the registry, that TP will fail.

ConversationSecurity:REG_SZ:{ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified:REG_SZ:{ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For a diagram of three TPs in a conversation, where the third TP can be invoked with a password that is already verified by the second TP, see Communication Between TPs. The following table shows the requirements for using password verification in a chain of TPs.

First TP (an invoking TP)	Second TP (invokable TP that confirms password and then invokes another TP)	Third and subsequent TPs (invokable TPs that invoke other TPs)
Does not need registry or environment variables.	**ConversationSecurity** setting must be YES.	**ConversationSecurity** setting must be YES.
Does not need registry or environment variables.	**AlreadyVerified** setting can be YES or NO.	**AlreadyVerified** setting must be YES.
Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies PROGRAM for the security type; as a result, the TP passes along the user identifier and password supplied in the symbolic destination name (or through calls(1)).	Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies SAME for the security type; as a result, after confirming the user identifier and password, the TP passes along the user identifier and an already-verified flag.	Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies SAME for the security type; as a result, the TP passes along the user identifier as received, along with the already-verified flag.

Note

1 Set_Conversation_Security_User_ID or Set_Conversation_Security_Password will overwrite the user identifier and password specified in the symbolic destination name.

If you set **AlreadyVerified** to NO, this TP cannot join in a chain of conversations where password verification is already done. (The exception to this is when **ConversationSecurity** is set to NO, in which case the TP could be the final TP in such a chain, since

it performs no checking.)

If you are configuring a TP that sometimes needs to confirm a password and sometimes accepts an already-verified flag, set AlreadyVerified to YES and configure the *UsernameX* variable appropriately. In this case, whenever the TP is invoked without the already-verified flag set, AlreadyVerified is ignored; verification is attempted with the user identifier and password configured for the TP.

The default for AlreadyVerified is NO. If you set AlreadyVerified to YES, make sure that ConversationSecurity is also set to YES.

Username1:REG_SZ:*Password1*

...

UsernameX:REG_SZ:*PasswordX*

Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can each be as many as 10 characters. Both parameters are case-sensitive.

This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the AlreadyVerified entry.

Example of Registry Entries for Windows 2000 or Windows NT

For an autostarted invokable TP called **BounceTP** and running as a service, the following registry entries might be added to a client running Windows 2000 or Windows NT. The entries would be added to **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services**, under the subkeys shown in bold type.

> **Note** In the following list, the parameters listed directly under the **BounceTP** key (such as **DisplayName** and **ErrorControl**) are service parameters created when TPSETUP or similar code is run to install the TP. These parameters should be created by TPSETUP or similar code; they should not be set manually. For more information about TPSETUP, see Sample CPI-C TPs in the SDK.

BounceTP
DisplayName:REG_SZ:BounceTP
ErrorControl:REG_DWORD:0x1
ImagePath:REG_EXPAND_SZ:c:\sna\system\bouncetp.exe
ObjectName:REG_SZ:LocalSystem
Start:REG_DWORD:0x3
Type:REG_DWORD:0x10

Linkage

OtherDependencies:REG_MULTI_SZ:SnaBase

Parameters

SNAServiceType:REG_DWORD:0x5
LocalLU:REG_SZ:JohnDoe
Parameters:REG_SZ:Arg1 Arg2 Arg3
Timeout:REG_DWORD:0x100
ConversationSecurity:REG_SZ:yes
AlreadyVerified:REG_SZ:no
JohnDoe:REG_SZ:SecretPassword

Clients Running Windows 98 or Windows 95

On clients running Microsoft® Windows® 98 or Microsoft® Windows® 95, invokable TPs are configured through the Windows 98 or Windows 95 registry.

> **Note** With Windows 98 or Windows 95, the recommended method for setting registry variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile INSTALL.C, the source code for TPSETUP, for the Windows 98 or Windows 95 environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample CPI-C TPs in the SDK.

The following table lists the registry entries used for the types of invokable TPs that can be run on Windows 98 or Windows 95 clients:

Type of TP	Location in registry	Possible registry entries
Autostarted invokable TP running as an application(1) on a Windows 98 or Windows 95 client	**HKEY_LOCAL_MACHINE** **SOFTWARE** **Microsoft** **SnaBase** **Parameters** **TPs** ***TPName*** Parameters	**SNAServiceType:**REG_DWORD:{ 0x5 | 0x6 } **PathName:**REG_EXPAND_SZ:*path* **LocalLU:**REG_SZ:*LUalias* **Parameters:**REG_SZ:*ParameterList* **TimeOut:**REG_DWORD:*number* **AcceptNames:**REG_SZ:*TPNameList* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }(2) ***Username1*:**REG_SZ:*Password1*(2) ... ***UsernameX*:**REG_SZ:*PasswordX*(2)
Operator-started invokable TP running as an application on a Windows 98 or Windows 95 client	**HKEY_LOCAL_MACHINE** **SOFTWARE** **Microsoft** **SnaBase** **Parameters** **TPs** ***TPName*** Parameters	**SNAServiceType:**REG_DWORD:0x1A **LocalLU:**REG_SZ:*LUalias* **TimeOut:**REG_DWORD:*number* **ConversationSecurity:**REG_SZ:{ YES | NO } **AlreadyVerified:**REG_SZ:{ YES | NO }(2) ***Username1*:**REG_SZ:*Password1*(2) ... ***UsernameX*:**REG_SZ:*PasswordX*(2)

Notes

1 Before an autostarted TP can be started as an application on a Windows 98 or Windows 95 client, the TPSTART program must be started. For more information, see Sample CPI-C TPs in the SDK.

2 **AlreadyVerified** and Username/Password entries are used only if **ConversationSecurity** is set to YES.

This section contains:

- Registry Entries for Clients Running Windows 98 or Windows 95
- Example of Registry Entries for Windows 98 and Windows 95

Registry Entries for Clients Running Windows 98 or Windows 95

The following list gives details about registry entries for clients running Windows 98 or Windows 95. For each TP type, the applicable variables and their locations are shown in Clients Running Windows 98 or Windows 95.

SNAServiceType:REG_DWORD:{ 0x5 | 0x6 | 0x1A }
Indicates the type of TP. Use a value of 0x5 for an autostarted queued TP, 0x6 for an autostarted nonqueued TP, and 0x1A for an operator-started TP.

PathName:REG_EXPAND_SZ:*path*
For an autostarted TP running as an application, specifies the path and file name of the TP. The data type of REG_EXPAND_SZ means that the path can contain an expandable data string; for example, %SystemRoot% represents the directory containing the Windows 98 or Windows 95 system files.

LocalLU:REG_SZ:*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters:REG_SZ:*ParameterList*
Lists parameters to be used by the TP. Separate parameters with spaces.

Timeout:REG_DWORD:*number*
Specifies the time, in milliseconds, that an Accept_Conversation will wait before timing out. Specify *number* in decimal; the registry editor converts this to hexadecimal before displaying it. The default is infinity (no limit).

AcceptNames:REG_SZ:*TPNameList*
With Windows 98 or Windows 95, this registry entry is used for autostarted TPs only. This entry lists additional names under which the invokable TP can be invoked. TP names should be separated with spaces. The default is none. If an invokable TP does not issue a Specify_Local_TP_Name for each name configured under AcceptNames in the registry, that TP will fail.

ConversationSecurity:REG_SZ:{ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified:REG_SZ:{ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For a diagram of three TPs in a conversation, where the third TP can be invoked with a password that is already verified by the second TP, see Communication Between TPs. The following table shows the requirements for using password verification in a chain of TPs.

First TP (an invoking TP)	Second TP (invokable TP that confirms password and then invokes another TP)	Third and subsequent TPs (invokable TPs that invoke other TPs)
Does not need registry or environment variables.	**ConversationSecurity** setting must be YES.	**ConversationSecurity** setting must be YES.
Does not need registry or environment variables.	**AlreadyVerified** setting can be YES or NO.	**AlreadyVerified** setting must be YES.
Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies PROGRAM for the security type; as a result, the TP passes along the user identifier and password supplied in the symbolic destination name (or through calls(1)).	Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies SAME for the security type; as a result, after confirming the user identifier and password, the TP passes along the user identifier and an already-verified flag.	Symbolic destination name or **Set_Conversation_Security_Type** in this TP specifies SAME for the security type; as a result, the TP passes along the user identifier as received, along with the already-verified flag.

Note

1. Set_Conversation_Security_User_ID or Set_Conversation_Security_Password will overwrite the user identifier and password specified in the symbolic destination name.

If you set **AlreadyVerified** to NO, this TP cannot join in a chain of conversations where password verification is already done. (The exception to this is when **ConversationSecurity** is set to NO, in which case the TP could be the final TP in such a chain, since it performs no checking.)

If you are configuring a TP that sometimes needs to confirm a password and sometimes accepts an already-verified flag, set AlreadyVerified to YES and configure the *UsernameX* variable appropriately. In this case, whenever the TP is invoked without the already-verified flag set, AlreadyVerified is ignored; verification is attempted with the user identifier and password configured for the TP.

The default for AlreadyVerified is NO. If you set AlreadyVerified to YES, make sure that ConversationSecurity is also set to YES.

***Username1*:**REG_SZ:*Password1*

...

***UsernameX*:**REG_SZ:*PasswordX*

Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can each be as many as 10 characters. Both parameters are case-sensitive.

This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the AlreadyVerified entry.

Example of Registry Entries for Windows 98 and Windows 95

For an autostarted invokable TP called **BounceTP** and running as a service, the following registry entries might be added to a Windows 98 or Windows 95 client. The entries would be added to **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft**, under the subkeys shown in bold type.

> **Note** In the following list, the parameters listed directly under the **BounceTP** key (such as **DisplayName** and **ErrorControl**) are service parameters created when TPSETUP or similar code is run to install the TP. These parameters should be created by TPSETUP or similar code; they should not be set manually. For more information about TPSETUP, see Sample CPI-C TPs in the SDK.

BounceTP
DisplayName:REG_SZ:BounceTP
ErrorControl:REG_DWORD:0x1
ImagePath:REG_EXPAND_SZ:c:\sna95\system\bouncetp.exe
ObjectName:REG_SZ:LocalSystem
Start:REG_DWORD:0x3
Type:REG_DWORD:0x10

Parameters

SNAServiceType:REG_DWORD:0x5
LocalLU:REG_SZ:JohnDoe
Parameters:REG_SZ:Arg1 Arg2 Arg3
Timeout:REG_DWORD:0x100
ConversationSecurity:REG_SZ:yes
AlreadyVerified:REG_SZ:no
JohnDoe:REG_SZ:SecretPassword

Clients Running Windows Version 3.x

On clients running Microsoft® Windows® version 3.*x*, invokable TPs are configured through entries in the WIN.INI file.

> **Note** With Windows version 3.x, the recommended method for setting environment variables for autostarted invokable TPs is to use the sample TP configuration program, TPSETUP. Compile INSTALL.C, the source code for TPSETUP, for the Windows version 3.x environment. When you write an installation program for autostarted invokable TPs, it is recommended that you add code similar to TPSETUP to the installation program. For information about TPSETUP, see Sample CPI-C TPs in the SDK.

The following table lists the section headings and environment variables used in the WIN.INI file for invokable TPs on clients running Windows version 3.*x*:

Type of TP	Section in WIN.INI listing TP names only	Section and possible environment variables defining TP
Autostarted invokable TP on a client running Windows version 3.*x*	**[SNAServerAutoTPs]** ***TPName****1*=*SectionName1* ... ***TPNameX***=*SectionNameX*	**[*SectionName1*]** **PathName**=*path* **LocalLU**=*LUalias* **Parameters**=*ParameterList* **TimeOut**=*number* **Queued**={ YES | NO } **AcceptNames**=*TPNameList* **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }(1) ***Username1***=*Password1*(1) ... ***UsernameX*=***PasswordX*(1)
Operator-started invokable TP on a client running Windows version 3.*x*	**[SNAServerAutoTPs]** ***TPNameN***=*SectionNameN* ... ***TPNameX***=*SectionNameX*	**[*SectionNameN*]** **LocalLU**=*Lualias* **TimeOut**=*number* **Queued**=OPERATOR **AcceptNames**=*TPNameList* **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }(1) ***Username1***=*Password1*(1) ... ***UsernameX*=***PasswordX*(1)

Note

1 AlreadyVerified and user name/password lines are used only if ConversationSecurity is set to YES.

This section contains:

- Environment Variables for Clients Running Windows Version 3.x
- Translating SNA Service TP Names to ASCII for WIN.INI
- Example of WIN.INI Lines for an Invokable TP

Environment Variables for Clients Running Windows Version 3.x

The following list shows the correct form for the sections and entries to add to the WIN.INI file for autostarted invokable TPs on a client running Windows version 3.*x*. The section headings are shown enclosed in square brackets; include the brackets when adding the section to the WIN.INI file.

For each TP type, the applicable variables and their locations are shown in Clients Running Windows Version 3.x.

[SNAServerAutoTPs]
TPNameX=*SectionNameX*
For all TPs. Associates *TPnameX* with *SectionNameX*. Additional lines can follow *TPNameX*=*SectionNameX*, each one using the same syntax to name a different TP and the section containing the information for that TP.

[*SectionName*]
Forms a section heading for entries applying to one TP; *SectionName* must match a section name listed under **[SNAServerAutoTPs]**.

PathName=*path*
Specifies the full path and file name of the executable file. The default is TPNAME.EXE.

LocalLU=*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters=*ParameterList*
Lists strings to be passed as command-line parameters for the TP. Separate parameters with spaces. The default is no parameters.

Timeout=*number*
Specifies the time in milliseconds that an Accept_Conversation will wait before timing out. The default is infinity (no limit).

Queued={ YES | NO }
Queued=OPERATOR
Specifies the type of TP: YES for an autostarted queued TP, NO for an autostarted nonqueued TP, or OPERATOR for an operator-started TP (which must always be queued). The default is YES.

AcceptNames=*TPNameList*
Lists additional names under which the invokable TP can be invoked. Separate TP names with spaces. The default is none. If an invokable TP does not issue a Specify_Local_TP_Name for each name configured under **AcceptNames** in the registry, that TP will fail.

ConversationSecurity={ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified={ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For detailed information about the AlreadyVerified variable, see the description for it under Registry Entries for Clients Running Windows 2000 or Windows NT.

The default is NO.

Username1=*Password1*
...
UsernameX=*PasswordX*
Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can have as many as 10 characters each. Both parameters are case-sensitive. This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the **AlreadyVerified** entry.

Translating SNA Service TP Names to ASCII for WIN.INI

For SNA service TPs on Host Integration Server or SNA Server clients running Windows version 3.*x*, the line naming the TP in the WIN.INI file must specify the TP name in ASCII. The following paragraphs tell how to convert a TP name to this form. The line should be placed in the **[SNAServerAutoTPs]** section of the file, as shown in Clients Running Windows Version 3.x.

An SNA service TP name is normally up to four bytes in length; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for WIN.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, a service TP name of 0x21 0xD7 0xD7 is equivalent to SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Example of WIN.INI Lines for an Invokable TP

For autostarted invokable TPs called **BounceTP** and **TestTP** on a client running Windows version 3.*x*, the following WIN.INI lines might be added:

```
[SNAServerAutoTPs]
BounceTP=bnceprms
TestTP=testprms

[bnceprms]
PathName=c:\sna\wbounce.exe
LocalLU=Eric
Parameters=/t
timeout=60000
queued=yes

[testprms]
PathName=c:\sna\testtp.exe
LocalLU=LU1
Parameters=/v
timeout=60000
queued=no
```

Clients Running OS/2

Host Integration Server does not include clients for OS/2 or support for developing CPI-C applications on OS/2. Earlier versions of SNA Server included SNA clients for OS/2 and support for developing CPI-C applications on OS/2.

On OS/2-based clients, invokable TPs are configured through entries in the SNA.INI file. The following table lists the section headings and environment variables used:

Type of TP	Section in SNA.INI listing TP names only	Section and possible environment variables defining TP
Autostarted invokable TP on a client running OS/2	**[SNAServerAutoTPs]** ***TPName1***=*SectionName1* ... ***TPNameX***=*SectionNameX*	**[*SectionName1*]** **PathName**=*path* **LocalLU**=*LUalias* **Parameters**=*ParameterList* **TimeOut**=*number* **Queued**={ YES | NO } **AcceptNames**=*TPNameList* **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }(1) ***Username1***=*Password1*[1] ... ***UsernameX***=*PasswordX*(1) **Environment**=*VariableList* **NewScreenGroup**={ 1 | 0 } **IconFile**=*path* **SessionType**=*number* **PgmControl**=*number* **InitXPos**=*number* **InitYPos**=*number* **InitXSize**=*number* **InitYSize**=*number*
Operator-started invokable TP on a client running OS/2	**[SNAServerAutoTPs]** ***TPNameN***=*SectionNameN* ... ***TPNameX***=*SectionNameX*	**[*SectionNameN*]** **LocalLU**=*LUalias* **TimeOut**=*number* **Queued**=OPERATOR **AcceptNames**=*TPNameList* **ConversationSecurity**={ YES | NO } **AlreadyVerified**={ YES | NO }(1) ***Username1***=*Password1*(1) ... ***UsernameX*=***PasswordX*(1)

Note

1 AlreadyVerified and user name/password lines are used only if ConversationSecurity is set to YES.

This section contains:

- Environment Variables for OS/2-Based Clients
- Translating SNA Service TP Names to ASCII for SNA.INI

Environment Variables for OS/2-Based Clients

The following list shows the correct form for the sections and entries to add to the SNA.INI file (located in the root SNA Server directory) for autostarted invokable TPs on an OS/2-based client. The section headings are shown enclosed in square brackets; include the brackets when adding the section to the SNA.INI file.

For each TP type, the applicable variables and their locations are shown in Clients Running OS/2.

[SNAServerAutoTPs]
TPNameX=*SectionNameX*
For all TPs. Associates *TPnameX* with *SectionNameX*. Additional lines may follow *TPNameX*=*SectionNameX*, each one using the same syntax to name a different TP and the section containing the information for that TP.

[*SectionName*]
Forms a section heading for entries applying to one TP; *SectionName* must match a section name listed under **[SNAServerAutoTPs]**.

PathName=*path*
Specifies the full path and file name of the executable file. The default is TPNAME.EXE.

LocalLU=*LUalias*
Specifies the alias of the local LU to be used when this TP is started on this computer.

Parameters=*ParameterList*
Lists strings to be passed as command line parameters for the TP. Separate parameters with spaces. The default is no parameters.

Timeout=*number*
Specifies the time, in milliseconds, that an Accept_Conversation will wait before timing out. The default is infinity (no limit).

Queued={ YES | NO }

Queued=OPERATOR
Specifies the type of TP: YES for an autostarted queued TP, NO for an autostarted nonqueued TP, or OPERATOR for an operator-started TP (which must always be queued). The default is YES.

AcceptNames=*TPNameList*
Lists additional names under which the invokable TP can be invoked. Separate TP names with spaces. The default is none. If an invokable TP does not issue a Specify_Local_TP_Name for each name configured under **AcceptNames** in the registry, that TP will fail.

ConversationSecurity={ YES | NO }
Indicates whether this TP supports conversation security. The default is NO.

AlreadyVerified={ YES | NO }
Indicates whether this TP can be invoked with a user identifier and password that have already been verified. **AlreadyVerified** is ignored if **ConversationSecurity** is set to NO.

For detailed information about the AlreadyVerified variable, see the description for it under Registry Entries for Clients Running Windows 2000 or Windows NT.

The default is NO.

Username1*=Password1*
...
UsernameX*=PasswordX*
Sets one or more user names and passwords to be compared with those sent by the invoking TP. The user name and password can have as many as 10 characters each. Both parameters are case-sensitive. This variable is ignored if conversation security is not activated or if the password has already been verified, as described for the **AlreadyVerified** entry.

Environment=*VariableList*
For an autostarted TP, lists the variables to be set in the TP's environment. Separate multiple variables with spaces.

NewScreenGroup={ 1 | 0 }
For an autostarted TP, specifies 1 to indicate that the TP runs in the foreground, or 0 to indicate that the TP runs in the background. The default is 1 (foreground).

IconFile=*path*
For an autostarted TP, specifies the full path and file name of the icon file. The default is no icon file.

SessionType=*number*
For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 0.

PgmControl=*number*
For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 32768.

InitXPos=*number*
For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 80.

InitYPos=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 80.

InitXSize=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 470.

InitYSize=*number*

For an autostarted TP, specifies a value used by OS/2; see the note following. The default is 330.

Note **SessionType**, **PgmControl**, **InitXPos**, **InitYPos**, **InitXSize**, and **InitYSize** are filled directly into the **STARTDATA** structure passed to the OS/2 **DosStartSession** call for the new session.

Translating SNA Service TP Names to ASCII for SNA.INI

For SNA service TPs on SNA Server clients running OS/2, the line naming the TP in the SNA.INI file must specify the TP name in ASCII. The following paragraphs tell how to convert a TP name to this form. The line should be placed in the **[SNAServerAutoTPs]** section of the file, as shown in Clients Running OS/2.

An SNA service TP name is normally up to four bytes in length; the first byte is a hexadecimal number in the range 0x00 to 0x3F, and the remaining bytes are EBCDIC characters. The first byte defines the function class of the TP. Therefore, to convert a service TP name to an ASCII form, convert the first byte as shown in the following table, and convert the EBCDIC values to ASCII letter equivalents.

First byte of TP name (hexadecimal number)	ASCII character equivalent for SNA.INI
0x07	DDM
0x20	DIA
0x21	SNAD
0x24	FS
0x30	PO
All others	UN

For example, a service TP name of 0x21 0xD7 0xD7 is equivalent to SNADPP (0x21 converts to SNAD and each 0xD7 converts to P).

Clients Running MS-DOS

Host Integration Server and SNA Server do not support invokable TPs on Microsoft® MS-DOS®-based clients.

Configuring Host Integration Server to Support TPs

The following topics describe how the Microsoft® Host Integration Server and the earlier Microsoft® SNA Server configuration works with invoking and invokable TPs.

This section contains:

Invoking TPs and the SNA Server Configuration

For an SNA server to support the beginning of the invoking process, the following parameters must be configured correctly:

- If the invoking TP specifies the LU alias that it uses (in a registry or environment variable), that LU alias must match a local APPC LU alias on the supporting SNA server. If the invoking TP does not specify a local LU alias, one of two methods for designating a default LU must be carried out on the supporting SNA server:
- Assign a default local APPC LU to the user or group that starts the invoking TP (that is, the user or group logged on at the system from which Initialize_Conversation is issued).

  —or—

  Designate one or more LUs as members of the default outgoing local APPC LU pool. SNA Server first attempts to determine the default local APPC LU of the user who started the TP, then attempts to assign an available LU from the default outgoing local APPC LU pool; if these attempts fail, SNA Server rejects the request.

- In most situations, the supporting SNA server must contain an appropriate connection to another system (host or peer). Sometimes, for testing purposes, the SNA server contains two local LUs paired together (for invoking and invokable TPs that are in the same domain); in this situation, a connection to a host or peer is not necessary.
- The partner LU alias specified in the symbolic destination name must match an LU alias that is paired with the local LU alias used by the invoking TP.

The preceding parameters support the beginning of the invoking process. For the invoking process to successfully complete, additional parameters must be configured as described in Invokable TPs and the SNA Server Configuration.

Invokable TPs and the SNA Server Configuration

For an SNA server to receive allocation requests from an invoking TP on another system and route those requests to an invokable TP, certain parameters must be configured correctly:

- The SNA server must have a connection to the system from which the invoking TP's request is sent.
- The SNA server must have a remote LU capable of receiving the incoming request. This remote LU can be configured either explicitly or implicitly.

  When configured explicitly, there is an explicit match between a remote LU alias on the SNA server and the alias of the LU that conveys the invoking TP's request.

  When configured implicitly, an implicit incoming remote LU (with its implicit incoming mode) is used. This means that several items must work together. First, the LU alias specified in the incoming request (the LU alias requested for the invokable TP) must match a local LU alias on the SNA server receiving the request. Second, the local LU on the server must have an implicit incoming remote LU assigned to it. The properties of the implicit incoming remote LU will be used for that LU-LU session. For more details about how an implicit incoming remote LU works, see the *Microsoft SNA Server Administration Guide*.

- Appropriate local LUs must be defined in the SNA server configuration. For descriptions of several ways to set up these local LUs, see Arranging TPs Within an SNA Network.

Arranging TPs Within an SNA Network

If your Host Integration Server or SNA Server installation contains multiple systems (clients and/or SNA servers), you can place a given invokable TP on more than one system. When an invoking request is received in such an installation, there can be a choice of systems on which to run the invokable TP. You can maintain specific control over this choice; alternatively, by following the instructions in TP Name Not Unique; Local LU Alias Unspecified, you can allow Host Integration Server or SNA Server to make the choice randomly to distribute the load.

You can maintain specific control over this choice of system by setting up invokable TPs with unique names, or by setting up each invokable TP to run only with a specific, unique LU alias. With this arrangement, the information provided by the invoking TP (in the symbolic destination name) specifies the system on which the invokable TP should run.

You can allow Host Integration Server or SNA Server to make the system choice randomly by setting the DloadMatchLocalFirst registry entry to NO, as described in the Installation and Configuration section of the *Microsoft Host Integration Server Guide* or the earlier *Microsoft SNA Server Reference*, and using invokable TPs that leave the local LU alias unspecified. Then, when an incoming request is received, it is routed randomly, rather than preferentially to the local server; in addition, no matter what LU alias is requested for the invokable TP, there cannot be a mismatch. SNA Server starts one instance of the requested TP, choosing randomly among the available systems.

The following topics describe some of the possible arrangements that can be made for running TPs.

This section contains:

- TP Name Unique for Each TP
- TP Name Not Unique; Local LU Alias Unique
- TP Name Not Unique; Local LU Alias Unspecified

TP Name Unique for Each TP

One way to specify the intended system where the invokable TP will run is to use a unique TP name for each invokable TP. In this arrangement, the invoking TP identifies the intended invokable TP (and system) simply by naming the TP. This makes it unnecessary for an invokable TP to specify any LU alias in registry or environment variables.

TP Name Not Unique; Local LU Alias Unique

Another way to specify the intended system where the invokable TP will run is to give the same name to multiple invokable TPs, but associate each TP with a unique local LU alias. To do this, configure each invokable TP (through registry or environment variables) to use a unique local LU alias. Then set up the invoking TPs so that each one is routed not only to the correct TP name but also to the correct partner LU alias for the intended invokable TP.

TP Name Not Unique; Local LU Alias Unspecified

If it does not matter on which system an invokable TP runs, use the same name for multiple invokable TPs and do not specify an LU alias in the registry or environment variables for the TPs. In this situation, there are no associated LU aliases in the list of available invokable TP names on an SNA server. Thus, a request received from an invoking TP cannot cause a mismatch on the LU alias, and will match according to the TP name.

In this situation, if you set the DloadMatchLocalFirst registry entry to NO, as described in the Installation and Configuration section of the *Microsoft Host Integration Server Guide* or the earlier *Microsoft SNA Server Reference*, the SNA server randomly routes the request to one of the available TPs. This spreads the processing load among multiple systems and provides hot backup (the ability to take systems online and offline without disrupting service).

Troubleshooting for Invokable TPs

If there are difficulties with starting an invokable TP, there may be a mismatch between the information for the invokable TP, the invoking TP, and/or LUs in the SNA Server configuration. That is, there may be a mismatch between the symbolic destination parameters, the registry or environment variables, and/or LU aliases specified in SNA Server Manager. For details about how to specify LU aliases in SNA Server Manager, see the Installation and Configuration section of the *Microsoft Host Integration Server Guide* or the earlier the *Microsoft SNA Server Administration Guide*.

Simplifying CPI-C Configuration

There are several features in SNA Server that can simplify configuration for CPI-C:

- The implicit incoming remote LU and the implicit incoming mode, which allow SNA Server to accept requests that arrive by unrecognized remote LUs and modes.
- The default local APPC LU and the default remote APPC LU, which allow LU aliases to be associated with user or group names, simplifying the routing of incoming requests and the configuration of client systems.
- The default outgoing local APPC LU pool, which allows LUs to be allocated dynamically to any invoking TP that does not specify a local LU.
- Automatic partnering, which automatically creates LU-LU pairs and assigns modes to the pairs.

For more information about these features, see the Installation and Configuration section of the *Microsoft Host Integration Server Guide* or the earlier *Microsoft SNA Server Administration Guide*.

Support for CPI-C Automatic Logon

This topic describes the support for automatic logon for CPI-C applications that is available in Microsoft® Host Integration Server 2000 and in Microsoft® SNA Server version 3.0 with Service Pack 1 or higher. This feature requires specific configuration by the network administrator: For more information on configuring this feature, see the Installation and Configuration section of the Host Integration Server Guide or the SNA Server online documentation.

The CPI-C application must be invoked on the LAN side from a client of Host Integration Server or SNA Server. The client must be logged into a Microsoft® Windows 2000 or Microsoft® Windows NT® domain, but the client application can be running on any operating system that supports the Host Integration Server or SNA Server CPI-C APIs (Windows 2000, Windows NT, Windows 98, Windows 95, Windows 3.*x* or OS/2).

To use this feature, the CPI-C client application is coded to use "program" level security, with a special hard-coded user name of MS$SAME and password of MS$SAME. When this session allocation flows from client to SNA server, Host Integration Server or SNA Server looks up the host account and password corresponding to the Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

In CPI-C, this is done in three separate function calls:

- Call the Set_Conversation_Security_Type function with the **conversation_security_type** parameter set to CM_SECURITY_PROGRAM.
- Call the Set_Conversation_Security_User_ID function with the **security_user_ID** parameter set to the MS$SAME string and the **security_user_ID_length** parameter set to 7.
- Call the Set_Conversation_Security_Password function with the **security_password** parameter set to the MS$SAME string and the **security_password_length** parameter set to 7.

CPI-C Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about the calls, extensions, and return codes that make up the CPI-C.

This section contains:

CPI-C Calls

This section describes the CPI-C calls. The following information is supplied for each call:

- The pseudonym for the call and the actual C function name.
- A definition of the call.
- A list of the parameters used by the call and the data type for each parameter. The prototype of each function is declared in the WINCPIC.H file.
- A description of each input and output parameter. The parameter names are pseudonyms and the actual names for these parameters are declared by the application program. The description includes the possible values of the parameter.
- The conversation state(s) in which the call can be issued.
- The state(s) to which the conversation can change upon return from the call. Conditions that do not cause a state change are not noted. For example, parameter checks and state checks do not cause a state change.
- Additional information describing the use of the call.

Data Types

The data types for the parameters supplied to and received from CPI-C are established as symbolic constants by **#define** statements in the WINCPIC.H file. For example, CM_INT32 represents **signed long int** and CM_PTR represents **far ***. Using symbolic constants improves the portability of CPI-C applications.

For ease of understanding, this reference presents the data types in absolute (not **#defined**) terms.

In writing applications, you should use the symbolic constants from the WINCPIC.H file.

Symbolic Constants

Most parameters supplied to and returned by CPI-C are 32-bit integers. To simplify coding, the values for these parameters are represented by meaningful symbolic constants, which are established by **#define** statements in the WINCPIC.H header file. For example, the value CM_MAPPED_CONVERSATION represents the integer 1. For the sake of readability, use only the symbolic constants when writing programs.

Strings

All strings are in ASCII format when passed across the CPI-C interface.

Validity of Output Parameters

The parameters returned by CPI-C are valid only if the CPI-C call is executed successfully, as indicated by a return code of CM_OK.

Accept_Conversation

The **Accept_Conversation** call (function name **cmaccp**) is issued by the invoked program to accept the incoming conversation and set certain conversation characteristics. For a list of initial conversation characteristics, see Initial Conversation Characteristics.

```
CM_ENTRY Accept_Conversation(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Returned parameter. Specifies the identifier for the conversation. It is used by subsequent CPI-C calls and is returned if the return code is either CM_OK or CM_OPERATION_INCOMPLETE. If *return_code* is CM_OPERATION_INCOMPLETE, the *conversation_ID* can be used by the application to wait for or cancel the conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; there is no incoming conversation (blocking mode only), or no local TP name has been set up.

CM_OPERATION_INCOMPLETE
Primary return code; a nonblocking operation has been started on the conversation but is not complete. The program can issue Wait_For_Conversation to wait for the operation to complete or Cancel_Conversation to cancel the operation and conversation.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in RESET state when **Accept_Conversation** is issued.

If the call is successful, the conversation changes to RECEIVE state. If the call fails, the state remains unchanged.

Remarks

Upon successful execution of this call, CPI-C generates an 8-byte conversation identifier. This identifier is a required parameter for all other CPI-C calls issued by the invoked program on this conversation.

Incoming conversations will be accepted according to the target TP name that they specify, which must match local TP names that have been set up. Local TP names can be set up by implementation-dependent methods, or by the program calling Specify_Local_TP_Name. In this way, a program can have more than one local TP name. The program can call Extract_TP_Name to discover the name specified on the incoming conversation.

The operation is performed in nonblocking mode if the program has called **Specify_Local_TP_Name** previously; otherwise it is performed in blocking mode.

Allocate

The **Allocate** call (function name **cmallc**) is issued by the invoking program to allocate a conversation with the partner program, using the current conversation characteristics. CPI-C can also allocate a session between the local LU and partner LU if one does not already exist.

```
CM_ENTRY Allocate(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the conversation identifier. The value of this parameter was returned by Initialize_Conversation.
return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.
CM_OPERATION_NOT_ACCEPTED
Primary return code; this value indicates that a previous operation on this conversation is incomplete.
CM_OPERATION_INCOMPLETE
Primary return code; a nonblocking operation has been started on the conversation but is not complete. The program can issue Wait_For_Conversation to wait for the operation to complete or Cancel_Conversation to cancel the operation and conversation.
CM_PARAMETER_ERROR
Primary return code; one of the following occurred:

- The mode name derived from the side information or set by Set_Mode_Name is not valid.
- The mode name is used by SNA service TPs; the invoking program does not have the authority to use this mode name. An example is SNASVCMG.
- The partner program derived from the side information is an SNA service TP; the local program does not have the privilege required to allocate a conversation to an SNA service TP.
- The partner program is a service TP, which participates in basic conversations, but the conversation is set to CM_MAPPED_CONVERSATION.
- The partner LU name derived from the side information or set by Set_Partner_LU_Name is not valid.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is not valid, or the address of a variable is invalid.
CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.
CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.
CM_UNSUCCESSFUL
Primary return code; the conversation's return-control characteristic is set to CM_IMMEDIATE and the local LU did not have an available contention-winner session.

The following return codes can be generated if the conversation's return-control type is set to CM_WHEN_SESSION_ALLOCATED.

CM_ALLOCATE_FAILURE_NO_RETRY
Primary return code; the conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.
CM_ALLOCATE_FAILURE_RETRY
Primary return code; the conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

State Changes

The conversation must be in INITIALIZE state when **Allocate** is issued.

State changes, summarized in the following table, are based on the value of the *return_code* parameter.

return_code	**New state**
CM_OK	SEND
CM_ALLOCATE_FAILURE_NO_RETRY	RESET
CM_ALLOCATE_FAILURE_RETRY	RESET
All others	No change

Remarks

The type of conversation allocated is based on the conversation type characteristic: mapped or basic.

When the conversation has been allocated by this call, the following conversation characteristics cannot be changed:

- Conversation type
- Mode name
- Partner LU name
- Partner program name
- Return control
- Synchronization level
- Conversation security
- User identifier
- Password

To send the allocation request immediately, the invoking program can issue Flush or Confirm immediately after **Allocate**. Otherwise, the allocate request accumulates with other data in the local LU's send buffer until the buffer is full.

By issuing **Confirm** after **Allocate**, the invoking program can immediately determine whether the allocation was successful (if the conversation synchronization level is set to CM_CONFIRM).

If the partner LU rejects the allocation request generated by **Allocate**, the error is returned to the invoking program on a subsequent call.

Cancel_Conversation

The **Cancel_Conversation** call (function name **cmcanc**) cancels any outstanding operation on a conversation (an operation returned with CM_OPERATION_INCOMPLETE) and the conversation itself.

```
CM_ENTRY Cancel_Conversation(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Returned parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in any state except RESET.

When the return code is CM_OK, the conversation state becomes RESET.

Remarks

Cancel_Conversation can be called while another operation is active for the specified *conversation_ID*. This allows an application to end any CPI-C action, but will terminate the conversation. This call can be issued regardless of the current application processing mode. Any outstanding operations will return with CM_DEALLOCATED_ABEND as the return code.

The conversation is terminated by a Deallocate with *deallocate_type* set to ABEND_SVC. No *log_data* is sent. The system may be unable to do this immediately, but any delay is transparent to the program.

> **Note** If Cancel_Conversation is called while there are outstanding Specify_Windows_Handle asynchronous calls, these calls are canceled. The return codes are set to canceled and a completion message is posted.

Confirm

The **Confirm** call (function name **cmcfm**) sends the contents of the local LU's send buffer and a confirmation request to the partner program and waits for confirmation. To avoid blocking for clients running Microsoft® Windows® version 3.*x*, use the Specify_Windows_Handle call. For Microsoft® Windows 2000, Microsoft® Windows NT®, Windows 98®, and Windows 95®, run a background thread for all CPI-C communications and preserve the foreground thread for user interface only.

```
CM_ENTRY Confirm(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *request_to_send_received,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

request_to_send_received
Returned parameter. Provides the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program issued Request_To_Send, which requests the local program to change the conversation to RECEIVE state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program did not issue **Request_To_Send**. This value is not relevant if *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully. The partner program issued the Confirmed call.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The local program attempted to use **Confirm** in a conversation with a synchronization level of CM_NONE. The synchronization level must be CM_CONFIRM.

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The conversation was not in SEND or SEND_PENDING state.
- The basic conversation for the local program was in SEND state, and the local program did not finish sending a logical record.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID

Primary return code; the user identifier or password specified in the allocation request is not accepted by the partner LU.

CM_SYNC LEVEL_NOT_SUPPORTED_PGM

Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED

Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY

Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY

Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_PURGING

Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued Send_Error. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY

Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY

Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND

Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The partner program terminated normally but did not deallocate the conversation before terminating.

CM_DEALLOCATED_ABEND_SVC

Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER

Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING

Primary return code; while in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data sent to the partner program may have been purged.

State Changes

The conversation can be in SEND or SEND_PENDING state when **Confirm** is issued.

State changes, summarized in the following table, are based on the value of the *return_code* parameter.

return_code	**New state**
CM_OK	
Call was issued in SEND state	No change
Call was issued in SEND_PENDING state	SEND

CM_PROGRAM_ERROR_PURGING	RECEIVE
CM_SVC_ERROR_PURGING	RECEIVE
CM_CONVERSATION_TYPE_MISMATCH	RESET
CM_PIP_NOT_SPECIFIED_CORRECTLY	RESET
CM_SECURITY_NOT_VALID	RESET
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	RESET
CM_TPN_NOT_RECOGNIZED	RESET
CM_TP_NOT_AVAILABLE_NO_RETRY	RESET
CM_TP_NOT_AVAILABLE_RETRY	RESET
CM_RESOURCE_FAILURE_NO_RETRY	RESET
CM_RESOURCE_FAILURE_RETRY	RESET
CM_DEALLOCATED_ABEND	RESET
CM_DEALLOCATED_ABEND_SVC	RESET
CM_DEALLOCATED_ABEND_TIMER	RESET
All others	No change

Remarks

In response to **Confirm**, the partner program normally issues Confirmed to confirm that it has received the data without error. (If the partner program encounters an error, it issues Send_Error or uses Deallocate to abnormally deallocate the conversation.)

The program can issue **Confirm** only if the conversation's synchronization level is CM_CONFIRM.

Confirm waits for a response from the partner program. A response is generated by one of the following CPI-C calls in the partner program:

Confirmed

Send_Error

Deallocate with the conversation's deallocate type set to CM_DEALLOCATE_ABEND

Confirmed

The **Confirmed** call (function name **cmcfmd**) replies to a confirmation request from the partner program. It informs the partner program that the local program has not detected an error in the received data. Because the program issuing the confirmation request waits for a confirmation, **Confirmed** synchronizes the processing of the two programs.

```
CM_ENTRY Confirmed(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation was not in CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state when the program issued this call.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in one of the following states when the program issues **Confirmed**:

CONFIRM

CONFIRM_SEND

CONFIRM_DEALLOCATE

The new state is determined by the old state—the state of the conversation when the local program issued **Confirmed**. The old state is indicated by the *status_received* value of the preceding Receive call. The following table summarizes the possible state changes when *return_code* is set to CM_OK.

Old state	New state
CONFIRM	RECEIVE
CONFIRM_SEND	SEND
CONFIRM_DEALLOCATE	RESET

Other return codes result in no state change.

Remarks

A confirmation request is issued by one of the following calls in the partner program:

- Confirm.
- Prepare_To_Receive if the prepare-to-receive type is set to CM_PREP_TO_RECEIVE_CONFIRM or to

CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM.

- Deallocate if the deallocate type is set to CM_DEALLOCATE_CONFIRM or to CM_DEALLOCATE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM.
- Send_Data under the following circumstances:

  The send type is set to CM_SEND_AND_CONFIRM.

  The send type is set to CM_SEND_AND_PREPARE_TO_RECEIVE and the prepare-to-receive type is set to CM_PREPARE_TO_RECEIVE_CONFIRM.

  The send type is set to CM_SEND_AND_PREPARE_TO_RECEIVE, the prepare-to-receive type is set to CM_PREPARE_TO_RECEIVE_SYNC_LEVEL, and the synchronization level is set to CM_CONFIRM.

  The send type is set to CM_SEND_AND_DEALLOCATE and the deallocate type is set to CM_DEALLOCATE_CONFIRM.

  The send type is set to CM_SEND_AND_DEALLOCATE, the deallocate type is set to CM_DEALLOCATE_SYNC_LEVEL, and the synchronization level is set to CM_CONFIRM.

A confirmation request is received by the local program through the *status_received* parameter of Receive. The local program can issue **Confirmed** only if the *status_received* parameter is set to one of the following values:

CM_CONFIRM_RECEIVED

CM_CONFIRM_SEND_RECEIVED

CM_CONFIRM_DEALLOC_RECEIVED

Convert_Incoming

The **Convert_Incoming** call (function name **cmcnvi**) converts a string of EBCDIC characters into ASCII. Note that the return conversion can be performed using **Convert_Outgoing**.

```
CM_ENTRY Convert_Incoming(
  unsigned char FAR *string,
  CM_INT32 FAR *string_length,
  CM_INT32 FAR *return_code
);
```

Parameters

string
Supplied parameter. Specifies the EBCDIC string to be converted. The string may contain any of the following characters:

- uppercase A-Z
- lowercase a-z
- 0-9
- the period (.)
- space characters
- the special characters < > + - ( ) & * ; : , ' ? / _= ".

string_length characters of this string will be replaced by ASCII equivalents.

string_length
Supplied parameter. Specifies the number of characters to be converted (1 - 32767).

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully and the *string* parameter now contains the converted ASCII string.

CM_OPERATION_NOT_ACCEPTED
Primary return code; the *string_length* parameter specified an invalid value.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state.

There is no state change.

Remarks

When data is being received in buffer format in a basic conversation, the data buffer may contain multiple logical records, each consisting of a two-byte length field (NN) followed by the data. The application must extract and convert each data string separately (excluding the length field value). The applications must not attempt to convert the whole buffer in one operation, because this will make the length field values invalid.

Convert_Outgoing

The **Convert_Outgoing** call (function name **cmcnvo**) converts a string of ASCII characters into EBCDIC. Note that the return conversion can be performed using **Convert_Incoming**.

```
CM_ENTRY Convert_Outgoing(
  unsigned char FAR *string,
  CM_INT32 FAR *string_length,
  CM_INT32 FAR *return_code
);
```

Parameters

string
Supplied parameter. Specifies the ASCII string to be converted. The string may contain any of the following characters:

- uppercase A-Z
- lowercase a-z
- 0-9
- the period (.)
- space characters
- the special characters < > + - ( ) & * ; : , ' ? / _= ".

string_length characters of this string will be replaced by EBCDIC equivalents.

string_length
Supplied parameter. Specifies the number of characters to be converted (1 - 32767).

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully and the *string* parameter now contains the converted EBCDIC string.

CM_OPERATION_NOT_ACCEPTED
Primary return code; the *string_length* parameter specified an invalid value.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state.

There is no state change.

Remarks

When data is being received in buffer format in a basic conversation, the data buffer may contain multiple logical records, each consisting of a two-byte length field (NN) followed by the data. The application must extract and convert each data string separately (excluding the length field value). The applications must not attempt to convert the whole buffer in one operation, because this will make the length field values invalid.

Deallocate

The **Deallocate** call (function name **cmdeal**) deallocates a conversation between two programs.

```
CM_ENTRY Deallocate(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully; the conversation is deallocated.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; the following state errors can occur when the deallocate type indicates a normal deallocation (CM_DEALLOCATE_SYNC_LEVEL, CM_DEALLOCATE_FLUSH, CM_DEALLOCATE_CONFIRM):

- The conversation is not in SEND or SEND_PENDING state.
- For a basic conversation, the conversation is in SEND state, but the program did not finish sending a logical record.

The following return codes can be returned when the *deallocate_type* is set to CM_DEALLOCATE_CONFIRM or to CM_DEALLOCATE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID
Primary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC LEVEL_NOT_SUPPORTED_PGM
Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED
Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_PURGING

Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued Send_Error. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY
Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued **Deallocate** with the type parameter set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote TP terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote TP.

CM_DEALLOCATED_ABEND_SVC
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING
Primary return code; while in SEND state, the partner program or partner LU issued **Send_Error** with the type parameter set to SVC. Data sent to the partner program may have been purged.

State Changes

Depending on the value of the conversation's deallocate type parameter (set by Set_Deallocate_Type), the conversation can be in one of the states indicated in the following table when the program issues **Deallocate**:

Deallocate type	**Allowed state**
CM_DEALLOCATE_FLUSH	SEND or SEND_PENDING
CM_DEALLOCATE_CONFIRM	SEND or SEND_PENDING
CM_DEALLOCATE_SYNC_LEVEL	SEND or SEND_PENDING
CM_DEALLOCATE_ABEND	Any except RESET

State changes, summarized in the following table, are based on the value of the *return_code* parameter.

return_code	**New state**
CM_OK	RESET
CM_PROGRAM_ERROR_PURGING	RECEIVE
CM_SVC_ERROR_PURGING	RECEIVE
CM_CONVERSATION_TYPE_MISMATCH	RESET
CM_PIP_NOT_SPECIFIED_CORRECTLY	RESET
CM_SECURITY_NOT_VALID	RESET

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	RESET
CM_TPN_NOT_RECOGNIZED	RESET
CM_TP_NOT_AVAILABLE_NO_RETRY	RESET
CM_TP_NOT_AVAILABLE_RETRY	RESET
CM_RESOURCE_FAILURE_NO_RETRY	RESET
CM_RESOURCE_FAILURE_RETRY	RESET
CM_DEALLOCATED_ABEND	RESET
CM_DEALLOCATED_ABEND_SVC	RESET
CM_DEALLOCATED_ABEND_TIMER	RESET
All others	No change

Remarks

Before deallocating the conversation, this call performs the equivalent of either the Flush or Confirmed call, depending on the current conversation synchronization level and deallocate type. The deallocate type is set by Set_Deallocate_Type.

The partner program receives the deallocation notification through one of the following parameters:

- status_received is CM_CONFIRM_DEALLOC_RECEIVED
- *return_code* is CM_DEALLOCATED_NORMAL
- *return_code* is CM_DEALLOCATED_ABEND

After this call has successfully executed, the *conversation_ID* is no longer valid.

For a basic conversation, if the conversation's deallocate type is set to CM_DEALLOCATE_ABEND and the log data length is greater than zero, the local LU writes the log data (specified by Set_Log_Data) to the local error log and to the partner LU.

After **Deallocate** has been executed, the log data length is set to zero and the log data is set to null.

Delete_CPIC_Side_Information

The **Delete_CPIC_Side_Information** call (function name **xcmdsi**) deletes an entry from the side information table in memory. The side information entry is identified through the symbolic destination name.

```
CM_ENTRY Delete_CPIC_Side_Information(
  unsigned char FAR *key_lock,
  unsigned char FAR *sym_dest_name,
  CM_INT32 FAR *return_code
);
```

Parameters

key_lock
Supplied parameter. This parameter is ignored.
sym_dest_name
Supplied parameter. Specifies the symbolic destination name of the entry to be deleted.
return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.
CM_PROGRAM_PARAMETER_CHECK
Primary return code; the *sym_dest_name* parameter specified a nonexistent side information entry.
CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The call is not associated with a conversation and can be in any state.

There is no state change.

Remarks

The side information entry is removed immediately from the side information table in memory.

While this call is being executed, any calls issued by other CPI-C applications that set or extract side information are suspended. These calls include the following:

Set_CPIC_Side_Information

Extract_CPIC_Side_Information

Initialize_Conversation

Extract_Conversation_Security_Type

The **Extract_Conversation_Security_Type** call (function name **xcecst**) returns the security type for a specified conversation.

```
CM_ENTRY Extract_Conversation_Security_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_security_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

conversation_security_type
Returned parameter. Specifies the information the partner LU requires to validate access to the invoked program. Possible values are:

CM_SECURITY_NONE
The invoked program uses no conversation security.

CM_SECURITY_PROGRAM
The invoked program uses conversation security and thus requires a user identifier and password.

CM_SECURITY_SAME
The invoked program, invoked with a valid user identifier and password, in turn invokes another program (as illustrated in Communication Between TPs). For example, assume that program A invokes program B with a valid user identifier and password, and program B in turn invokes program C. If program B specifies the value CM_SECURITY_SAME, CPI-C sends the LU for program C, the user identifier from program A, and an already-verified indicator. This indicator tells program C not to require the password (if program C is configured to accept an already-verified indicator).

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid, or the address of a variable is invalid.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Extract_Conversation_Security_User_ID

The **Extract_Conversation_Security_User_ID** call (function name **cmecsu**) returns the user identifier being used in a specified conversation.

```
CM_ENTRY Extract_Conversation_Security_User_ID(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *security_user_ID,
  CM_INT32 FAR *security_user_ID_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

security_user_ID
Returned parameter. Specifies the user identifier that was used to establish the conversation.

security_user_ID_length
Returned parameter. Specifies the length of *security_user_ID*.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

The *security_user_ID* value is not padded with spaces. It is meaningful only up to *security_user_ID_length*.

Extract_Conversation_State

The **Extract_Conversation_State** call (function name **cmecs**) returns the state of the specified conversation.

```
CM_ENTRY Extract_Conversation_State(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_state,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

conversation_state
Returned parameter. Specifies the conversation state. Possible values are:

CM_INITIALIZE_STATE

CM_SEND_STATE

CM_RECEIVE_STATE

CM_SEND_PENDING_STATE

CM_CONFIRM_STATE

CM_CONFIRM_SEND_STATE

CM_CONFIRM_DEALLOCATE_STATE

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Extract_Conversation_Type

The **Extract_Conversation_Type** call (function name **cmect**) returns the conversation type—mapped or basic—of the specified conversation.

```
CM_ENTRY Extract_Conversation_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

conversation_type
Returned parameter. Specifies the conversation type. Possible values are:

CM_BASIC_CONVERSATION

CM_MAPPED_CONVERSATION

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Extract_CPIC_Side_Information

The **Extract_CPIC_Side_Information** call (function name **xcmesi**) returns the side information for an entry number or symbolic destination name.

```
CM_ENTRY Extract_CPIC_Side_Information(
  CM_INT32 FAR *entry_number,
  unsigned char FAR *sym_dest_name,
  SIDE_INFO FAR *side_info_entry,
  CM_INT32 FAR *side_info_entry_length,
  CM_INT32 FAR *return_code
);
```

Parameters

entry_number
Supplied parameter. Specifies the number (index) of the side information entry to be returned. The first entry is 1.

The program can look up the side information entry by the symbolic destination name instead. To accomplish this, set the entry number to zero.

sym_dest_name
Supplied parameter. Specifies the symbolic destination name to search for.

If *entry_number* is set to a number greater than zero, this parameter is ignored.

side_info_entry
Returned parameter. Specifies the side information entry. For a detailed explanation of the side information entry, see Set_CPIC_Side_Information.

Each field in the side information structure is left-aligned and padded with spaces on the right as necessary.

side_info_entry_length
Supplied parameter. Always 124.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The *entry_number* specified a number larger than the maximum number of entries in the side information table or a number that is less than zero.
- The *sym_dest_name* parameter is invalid and *entry_number* is set to zero.
- The *side_info_entry_length* parameter is not set to 124.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

This call is not associated with a conversation and can be in any state.

There is no state change.

Remarks

The security password is never returned. If the security user identifier in the side information is not set, the security user identifier field is returned as all spaces.

Extract_Mode_Name

The **Extract_Mode_Name** call (function name **cmemn**) returns the mode name and mode name length for a specified conversation.

```
CM_ENTRY Extract_Mode_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *mode_name,
  CM_INT32 FAR *mode_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

mode_name
Returned parameter. Specifies the starting address of the mode name.

mode_name_length
Returned parameter. Specifies the length of the mode name.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Extract_Partner_LU_Name

The **Extract_Partner_LU_Name** call (function name **cmepln**) returns the partner LU name and partner LU name length for a specified conversation. This can be an alias name of up to eight bytes or a fully qualified network name of up to 17 bytes.

```
CM_ENTRY Extract_Partner_LU_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *partner_LU_name,
  CM_INT32 FAR *partner_LU_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

partner_LU_name
Returned parameter. Specifies the variable containing the partner LU name. (The program must supply a pointer to a suitable variable.)

partner_LU_name_length
Returned parameter. Specifies the length of the partner LU name.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

An invokable CPI-C TP will only receive the fully qualified network name (FQLU) upon successful completion of this function call. An invokable CPI-C TP is unable to retrieve the alias name using this call.

Extract_Sync_Level

The **Extract_Sync_Level** call (function name **cmesl**) returns the synchronization level for a specified conversation.

```
CM_ENTRY Extract_Sync_Level(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *sync_level,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

sync_level
Returned parameter. Indicates the synchronization level of the conversation. Possible values are:

CM_NONE
The programs will not perform confirmation processing.

CM_CONFIRM
The programs can perform confirmation processing.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Extract_TP_Name

The **Extract_TP_Name** call (function name **cmetpn**) returns the *TP_name* characteristic.

```
CM_ENTRY Extract_TP_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *TP_name,
  CM_INT32 FAR *TP_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

TP_name
Returned parameter. Specifies the variable containing the TP name.

TP_name_length
Returned parameter. Specifies the length of the TP name.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

For an invoking program, the *TP_name* characteristic is the value in the side information referenced in the *sym_dest_name* parameter of the Initialize_Conversation call. For an invokable program, it is the name specified in the conversation startup request (which will have been matched with a name specified locally or in a Specify_Local_TP_Name call), and will therefore be the same as the *TP_name* characteristic of the partner program.

The name returned can be up to 64 bytes in length.

Flush

The **Flush** call (function name **cmflus**) sends the contents of the local LU's send buffer to the partner LU (and program). If the send buffer is empty, no action takes place.

```
CM_ENTRY Flush(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation was not in SEND or SEND_PENDING state when the program issued this call.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in SEND or SEND_PENDING state.

If the call completes successfully, (*return_code* is CM_OK), the conversation is in SEND state.

Other return codes result in no state change.

Remarks

Data processed by Send_Data accumulates in the local LU's send buffer until one of the following happens:

- The local program issues the Flush call or other call that flushes the LU's send buffer. (Some send types, set by Set_Send_Type, include flush functionality.)
- The buffer is full.

The allocation request generated by Allocate and error information generated by Send_Error are also buffered.

Initialize_Conversation

The **Initialize_Conversation** call (function name **cminit**) is issued by the invoking program to obtain an 8-byte conversation identifier and to set the initial values for the conversation's characteristics.

```
CM_ENTRY Initialize_Conversation(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *sym_dest_name,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Returned parameter. Specifies the identifier for the conversation. It is used by subsequent CPI-C calls.

sym_dest_name
Supplied parameter. Specifies the symbolic destination name—the name associated with a side information entry loaded from the configuration file or defined by Set_CPIC_Side_Information calls.

This parameter is an 8-byte ASCII character string. The allowed characters are as follows:

- Uppercase letters
- Numerals 0 through 9

This parameter can also be set to eight spaces. In this case, the invoking program must issue the following calls before issuing Allocate:

Set_Mode_Name

Set Partner_LU_Name

Set_TP_Name

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *sym_dest_name* does not match a symbolic destination name in the side information table and is not a space.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation is in RESET state.

If the *return_code* is CM_OK, the conversation changes to INITIALIZE state. For other return codes, the conversation state remains unchanged.

Remarks

The initial values are CPI-C defaults or are derived from side information associated with the symbolic destination name. For more information about initial values and side information, see Initial Conversation Characteristics and Side Information.

Initial values can be changed by the **Set**_ calls.

If the side information contains an invalid value or a **Set**_ call sets a conversation characteristic to an invalid value, the error is returned on the **Allocate** call.

If a CPIC application attempts to invoke more than one concurrent conversation, only a single Local APPC LU is used by all conversations. This prevents concurrent conversations across two or more dependent LU6.2 LU's, causing subsequent Initialize_Conversation (CMALLC)calls to wait for the first conversation to be deallocated.

If the CPIC application needs to invoke more than one concurrent conversation, independent LU6.2 must be used between

Microsoft® Host Integration Server or Microsoft® SNA Server and the remote system.

Upon successful execution of this call, CPI-C generates a conversation identifier. This identifier is a required parameter for all other CPI-C calls issued for this conversation by the invoking program.

Under normal circumstances, a CPI-C application cannot invoke two concurrent conversations using two different Local APPC LUs. A registry key is available that when set forces CPI-C to issue a new TP_STARTED verb on every Initialize_Conversation (cminit) call. This is necessary to force APPC resource location for each call. The registry key that must be defined to force this behavior is the following:

\HKLM\CurrentControlSet\Services\SnaBase\Parameters\Client\GETNEWTPID

Prepare_To_Receive

The **Prepare_To_Receive** call (function name **cmptr**) changes the state of the conversation for the local program from SEND to RECEIVE.

```
CM_ENTRY Prepare_To_Receive(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The conversation state is not SEND or SEND_PENDING.
- For a basic conversation, the conversation is in SEND state. However, the program did not finish sending a logical record.

These return codes can occur if the conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_CONFIRM or if the prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID
Primary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM
Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED
Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_PURGING
Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued Send_Error. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY
Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote TP terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote TP.

CM_DEALLOCATED_ABEND_SVC
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING
Primary return code; while in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data sent to the partner program may have been purged.

State Changes

The conversation can be in SEND or SEND_PENDING state.

State changes, summarized in the following table, are based on the value of the *return_code* parameter.

return_code	**New state**
CM_OK	RECEIVE
CM_PROGRAM_ERROR_PURGING	RECEIVE
CM_SVC_ERROR_PURGING	RECEIVE
CM_CONVERSATION_TYPE_MISMATCH	RESET
CM_PIP_NOT_SPECIFIED_CORRECTLY	RESET
CM_SECURITY_NOT_VALID	RESET
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	RESET
CM_TPN_NOT_RECOGNIZED	RESET
CM_TP_NOT_AVAILABLE_NO_RETRY	RESET
CM_TP_NOT_AVAILABLE_RETRY	RESET
CM_DEALLOCATED_ABEND	RESET
CM_RESOURCE_FAILURE_NO_RETRY	RESET

CM_RESOURCE_FAILURE_RETRY	RESET
CM_DEALLOCATED_ABEND_SVC	RESET
CM_DEALLOCATED_ABEND_TIMER	RESET
All others	No change

Before changing the conversation state, this call performs the equivalent of one of the following:

- The Flush call, sending the contents of the local LU's send buffer to the partner LU and program, if either of the following conditions is true:

  The conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_FLUSH.

  The conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_NONE.

- The Confirm call, sending the contents of the local LU's send buffer and a confirmation request to the partner program, if either of the following conditions is true:

  The conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_CONFIRM.

  The conversation's prepare-to-receive type is set to CM_PREP_TO_RECEIVE_SYNC_LEVEL and the conversation's synchronization level is set to CM_CONFIRM.

The prepare-to-receive type is set by Set_Prepare_To_Receive_Type; the synchronization level is set by Set_Sync_Level.

The conversation cannot change to SEND or SEND_PENDING for the partner program until the partner program receives one of the following values through the *status_received* parameter of the Receive call:

- CM_SEND_RECEIVED
- CM_CONFIRM_SEND_RECEIVED and replies with the Confirmed or Send_Error call

Remarks

After this call has successfully executed, the local program can receive data.

Receive

The **Receive** call (function name **cmrcv**) receives any data that is currently available from the partner program. To avoid blocking for clients running Microsoft® Windows® version 3.*x*, use the Specify_Windows_Handle call. For Microsoft® Windows 2000, Microsoft® Windows NT®, Windows 98®, and Windows 95®, run a background thread for all CPI-C communications and preserve the foreground thread for user interface only.

```
CM_ENTRY Receive(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *buffer,
  CM_INT32 FAR *requested_length,
  CM_INT32 FAR *data_received,
  CM_INT32 FAR *received_length,
  CM_INT32 FAR *status_received,
  CM_INT32 FAR *request_to_send_received,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

buffer
Returned parameter. Specifies the address of the buffer to contain the data received by the local program.

The buffer contains data if the following conditions are true:

- The *data_received* parameter is set to a value other than CM_NO_DATA_RECEIVED.
- The *return_code* parameter is set to CM_OK or to CM_DEALLOCATED_NORMAL.

requested_length
Supplied parameter. Indicates the maximum number of bytes of data the local program is to receive. The range is from 0 through 32767.

data_received
Returned parameter. Indicates whether the program received data. These codes are not relevant unless *return_code* is set to CM_OK or CM_DEALLOCATED_NORMAL. Possible values are listed following the Parameters section.

received_length
Returned parameter. Indicates the number of bytes of data the local program received on this **Receive** call. If *return_code* or *data_received* indicates that the program received no data, this number is not relevant.

status_received
Returned parameter. Indicates changes in the status of the conversation. These codes are not relevant unless *return_code* is set to CM_OK. Possible values are listed following the Parameters section.

request_to_send_received
Returned parameter. Specifies the request-to-send-received indicator. Possible values are listed following the Parameters section.

return_code
The code returned from this call. Possible values are listed following the Parameters section.

Values returned in the data_received parameter

CM_DATA_RECEIVED
Can be returned for a basic conversation if the conversation's fill characteristic is set to CM_FILL_BUFFER, indicating that the program is receiving data independent of its logical format. The local program received data until *requested_length* or end of data was reached.

The end of the data is indicated by either a change to another conversation state, based on the *return_code*, *status_received*, and *data_received* parameters, or an error condition. If the conversation's receive type is set to CM_RECEIVE_IMMEDIATE, the data received can be less than *requested_length* if a smaller amount of data has arrived from the partner program.

CM_COMPLETE_DATA_RECEIVED
In a mapped conversation, indicates that the local program has received a complete data record or the last part of a data record.

In a basic conversation with the fill characteristic set to CM_FILL_LL, this value indicates that the local program has received a

complete logical record or the end of a logical record.

CM_INCOMPLETE_DATA_RECEIVED
In a mapped conversation, indicates that the local program has received an incomplete data record. The *requested_length* parameter specified a value less than the length of the data record (or less than the remainder of the data record if this is not the first **Receive** to read the record). The amount of data received is equal to the *requested_length* parameter.

In a basic conversation with the fill characteristic set to CM_FILL_LL, this value indicates that the local program has received an incomplete logical record. The amount of data received is equal to the *requested_length* parameter. (If the received data was truncated, the length of the data will be less than *requested_length*.)

Upon receiving this value, the local program normally reissues **Receive** to receive the next part of the record.

CM_NO_DATA_RECEIVED
The program did not receive data.

Note that if the *return_code* parameter is set to CM_OK, status information may be available through the *status_received* parameter.

Values returned in the status_received parameter

CM_NO_STATUS_RECEIVED
No conversation status change was received on this call.
CM_SEND_RECEIVED
Indicates, for the partner program, that the conversation has entered RECEIVE state. For the local program, the conversation is now in SEND state if no data was received on this call, or SEND_PENDING state if data was received on this call.

Upon receiving this value, the local program normally uses Send_Data to begin sending data.

CM_CONFIRM_DEALLOC_RECEIVED
Indicates that the partner program issued Deallocate with confirmation requested. For the local program the conversation is now in CONFIRM_DEALLOCATE state.

Upon receiving this value, the local program normally issues the Confirmed call.

CM_CONFIRM_RECEIVED
Indicates that the partner program issued the Confirm call. For the local program, the conversation is in CONFIRM state.

Upon receiving this value, the local program normally issues the **Confirmed** call.

CM_CONFIRM_SEND_RECEIVED
Indicates, for the partner program, that the conversation has entered RECEIVE state and a request for confirmation has been received by the local program. For the local program, the conversation is now in CONFIRM_SEND state.

The program normally responds by issuing the **Confirmed** call. Upon successful execution of the **Confirmed** call, the conversation changes to SEND state for the local program.

Values returned in the request_to_send_received parameter

CM_REQ_TO_SEND_RECEIVED
The partner program issued the Request_To_Send call, which requests the local program to change the conversation to RECEIVE state.
CM_REQ_TO_SEND_NOT_RECEIVED
The partner program did not issue the **Request_To_Send** call. This value is not relevant if the *return_code* parameter is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK.

Values returned in the Return_Code parameter

CM_OK
Primary return code; the call executed successfully.
CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.
CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.
CM_UNSUCCESSFUL
Primary return code; the receive type is set to CM_RECEIVE_IMMEDIATE and no data is immediately available from the partner program.

CM_DEALLOCATED_NORMAL
Primary return code; the conversation has been deallocated normally. The partner program issued Deallocate with the conversation's deallocate type set to CM_DEALLOCATE_FLUSH or CM_DEALLOCATE_SYNC_LEVEL with the synchronization level of the conversation specified as CM_NONE.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *requested_length* is out of range (greater than 32767).

If the program receives this return code, the other returned parameters are not valid.

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The receive type is set to CM_RECEIVE_AND_WAIT and the conversation state is not RECEIVE, SEND, or SEND_PENDING.
- The receive type is set to CM_RECEIVE_IMMEDIATE and the conversation state is not RECEIVE.
- In a basic conversation, the conversation is in SEND state, the receive type is set to CM_RECEIVE_AND_WAIT, and the program did not finish sending a logical record.

If the program receives this return code, the other returned parameters are not valid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID
Primary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM
Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED
Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_NO_TRUNC
Primary return code; while in SEND state or in SEND_PENDING state with the error direction set to CM_SEND_ERROR, the partner program issued Send_Error. Data was not truncated.

CM_PROGRAM_ERROR_PURGING
Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued **Send_Error**. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY
Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote TP terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote TP.

CM_DEALLOCATED_ABEND_SVC
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING
Primary return code; while in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data sent to the partner program may have been purged.

CM_SVC_ERROR_NO_TRUNC
Primary return code; while in SEND state, the partner program or partner LU issued **Send_Error** with the type parameter set to SVC. Data sent to the partner program may have been purged.

CM_PROGRAM_ERROR_TRUNC
Primary return code; in SEND state, before finishing sending a complete logical record, the partner program issued **Send_Error**. The local program may have received the first part of the logical record through a **Receive** call.

CM_SVC_ERROR_TRUNC
Primary return code; while in RECEIVE or CONFIRM state, the partner program or partner LU issued **Send_Error** with the type parameter set to SVC before it finished sending a complete logical record. The local program may have received the first part of the logical record.

State Changes

The conversation can be in RECEIVE, SEND, or SEND_PENDING state.

If *receive_type* is set to CM_RECEIVE_IMMEDIATE, the conversation must be in RECEIVE state.

Issuing **Receive** while the conversation is in SEND or SEND_PENDING state causes the local LU to send the information in its send buffer and a send indicator to the partner program. Based on *data_received* and *status_received* the conversation can change to RECEIVE state for the local program.

The new conversation state is determined by:

- The state the conversation is in when the program issues the call.
- The *return_code* parameter.
- The data_received and status_received parameters.

If no data is currently available and the receive type (set by Set_Receive_Type) is set to CM_RECEIVE_AND_WAIT, the local program waits for data to arrive. If the receive type is set to CM_RECEIVE_IMMEDIATE, the local program does not wait.

The process for receiving data is as follows:

- The local program issues a Receive call until it finishes receiving a complete unit of data. The local program may need to issue Receive several times to receive a complete unit of data. The *data_received* parameter indicates whether the receipt of data is finished.

  The data received can be:

  One data record transmitted in a mapped conversation.

  One logical record transmitted in a basic conversation with the conversation's fill characteristic set to CM_FILL_LL.

A buffer of data received independent of its logical-record format in a basic conversation with the fill characteristic set to CM_FILL_BUFFER.

When a complete unit of data has been received, the local program can manipulate it.

- The local program determines the next action to take based on the control information received through *status_received*. The local program may have to reissue **Receive** to receive the control information.

The conversation type is set by Set_Conversation_Type; the fill characteristic is set by Set_Fill.

The following table summarizes the state changes that can occur when **Receive** is issued with the conversation in RECEIVE state and *return_code* is CM_OK.

data_received	***status_received***	**New state**
CM_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	No change
CM_COMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	No change
CM_INCOMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	SEND_PENDING
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	SEND

If *return_code* is set to CM_UNSUCCESSFUL, meaning that the *receive_type* is set to CM_RECEIVE_IMMEDIATE and no data is available, there is no state change.

The following table summarizes the state changes that can occur when **Receive** is issued with the conversation in SEND state and *return_code* is CM_OK.

data_received	***status_received***	**New state**
CM_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	RECEIVE
CM_COMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	RECEIVE
CM_INCOMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	SEND_PENDING
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	No change

The following table summarizes the state changes that can occur when **Receive** is issued with the conversation in SEND_PENDING state and *return_code* is CM_OK.

data_received	***status_received***	**New state**
CM_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	RECEIVE
CM_COMPLETE_DATA_RECEIVED	CM_NO_STATUS_RECEIVED	RECEIVE
CM_INCOMPLETE_DATA_RECEIVED	CM_SEND_RECEIVED	No change
CM_NO_DATA_RECEIVED	CM_SEND_RECEIVED	SEND

The following topics summarize state changes that can occur when **Receive** is issued in any allowed state.

Confirmation

The following table summarizes state changes that occur under the following conditions:

- The return_code parameter is CM_OK.
- The *data_received* parameter is set to CM_DATA_RECEIVED, CM_COMPLETE_DATA_RECEIVED, or CM_NO_DATA_RECEIVED.
- The *status_received* parameter indicates a change to a CONFIRM state.

status_received	**New state**
CM_CONFIRM_DEALLOC_RECEIVED	CONFIRM_DEALLOCATE
CM_CONFIRM_SEND_RECEIVED	CONFIRM_SEND
CM_CONFIRM_RECEIVED	CONFIRM

Normal Deallocation

If *return_code* is set to CM_DEALLOCATED_NORMAL, the conversation changes to RESET state.

ABEND

The following ABEND conditions, indicated by *return_code*, cause the conversation to change to RESET state:

CM_CONVERSATION_TYPE_MISMATCH

CM_PIP_NOT_SPECIFIED_CORRECTLY

CM_SECURITY_NOT_VALID

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

CM_SVC_ERROR_TRUNC

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

Errors

The following table summarizes state changes that occur when a data transmission error is encountered.

return_code	**Old state**	**New state**
CM_PROGRAM_ERROR_PURGING	RECEIVE	No change
CM_PROGRAM_ERROR_NO_TRUNC	RECEIVE	No change
CM_SVC_ERROR_PURGING	SEND	RECEIVE
CM_SVC_ERROR_NO_TRUNC	SEND_PENDING	RECEIVE

If the partner program truncates a logical record, the local program receives notification of the truncation through *return_code* on the next **Receive** call.

If a program issues **Receive** with *requested_length* set to zero, the call is executed as usual. However, *data_received* and *status_received* are not set on the same **Receive** call. (One exception to this situation is the null record sent over a mapped conversation, described in the next paragraph.)

In a mapped conversation in which data is available from the partner program, *data_received* is set to CM_INCOMPLETE_DATA_RECEIVED. If a null record is available (*send_length* in the Send_Data call issued by the partner program is set to zero), *data_received* is set to CM_COMPLETE_RECORD_RECEIVED with *received_length* set to zero.

In a basic conversation in which data is available and the fill characteristic is set to CM_FILL_LL, *data_received* is set to CM_INCOMPLETE_DATA_RECEIVED. If the fill characteristic is set to CM_FILL_BUFFER, *data_received* is set to CM_DATA_RECEIVED.

The LU does not automatically perform any conversion between EBCDIC and ASCII on the received string of data before putting it in *buffer*. If necessary, the program can use the CSV CONVERT to translate a string from one character set to the other.

Request_To_Send

The **Request_To_Send** call (function name **cmrts**) notifies the partner program that the local program wants to send data.

```
CM_ENTRY Request_To_Send(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in the RECEIVE, SEND, SEND_PENDING, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any of the following states: RECEIVE, SEND, SEND_PENDING, CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE.

There is no state change.

In response to this request, the partner program can change the conversation to RECEIVE state by issuing one of the following calls:

- Receive with receive type set to CM_RECEIVE_AND_WAIT
- Prepare_To_Receive
- **Send_Data** with send type set to CM_SEND_AND_PREP_TO_RECEIVE

The partner program can also ignore the request to send.

The conversation state changes to SEND for the local program when the local program receives one of the following values through the *status_received* parameter of a subsequent **Receive** call:

- CM_SEND_RECEIVED
- CM_CONFIRM_SEND_RECEIVED and the local program replies with a Confirmed call

Remarks

The request-to-send notification is received by the partner program through the *request_to_send_received* parameter of the following calls:

Confirmed

Receive

Send_Data

Send_Error

Test_Request_To_Send_Received_sna_Test_Request_To_Send_Received_cpic

Request-to-send notification is sent to the partner program immediately; CPI-C does not wait until the send buffer fills up or is flushed. Consequently, the request-to-send notification can arrive out of sequence. For example, if the local program is in SEND state and issues the Prepare_To_Receive call followed by the **Request_To_Send** call, the partner program, in RECEIVE state, can receive the request-to-send notification before it receives the send notification. For this reason, *request_to_send* can be reported to a program through the Receive call.

Upon receiving a request-to-send notification, the partner LU retains the notification until the partner issues a call that returns *request_to_send_received*. The LU keeps only one request-to-send notification per conversation. Thus the local program can issue more **Request_To_Send** calls than are explicitly handled by the partner TP.

Send_Data

The **Send_Data** call (function name **cmsend**) puts data in the local LU's send buffer for transmission to the partner program.

```
CM_ENTRY Send_Data(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *buffer,
  CM_INT32 FAR *send_length,
  CM_INT32 FAR *request_to_send_received,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

buffer
Supplied parameter. Specifies the address of the buffer containing the data to be put in the local LU's send buffer.

send_length
Supplied parameter. Specifies the number of bytes of data to be put in the local LU's send buffer. The range is from 0 through 32767.

For a mapped conversation, if *send_length* is set to zero, a null data record is sent to the partner program.

For a basic conversation, if *send_length* is set to zero, no data is sent. The *buffer* parameter is not relevant. However, the other parameters are processed.

request_to_send_received
Returned parameter. Is the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program issued the Request_To_Send call, which requests the local program to change the conversation to RECEIVE state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program did not issue the **Request_To_Send** call. This value is not relevant if *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_PROGRAM_STATE_CHECK.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *send_length* is out of range (greater than 32767).
- This is a basic conversation and the first two bytes of *buffer* contain an invalid logical record length (0x0000, 0x0001, 0x8000, or 0x8001).

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The conversation state is not SEND or SEND_PENDING.
- The basic conversation is in SEND state and *send_type* is set to CM_SEND_AND_CONFIRM,

CM_SEND_AND_DEALLOCATE, or CM_SEND_AND_PREP_TO_RECEIVE. However, the data does not end on a logical record boundary. This condition is allowed only when *deallocate_type* is set to CM_DEALLOCATE_ABEND and the *send_type* is set to CM_SEND_AND_DEALLOCATE.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID
Primary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM
Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED
Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_PURGING
Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued Send_Error. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY
Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND, or the remote LU did so because of a remote program abnormal-ending condition. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote TP terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote TP.

CM_DEALLOCATED_ABEND_SVC
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type

parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING

Primary return code; while in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data sent to the partner program may have been purged.

State Changes

The conversation must be in SEND or SEND_PENDING state when the program issues this call.

The following tables summarize state changes that are possible when *return_code* is set to CM_OK.

send_type	**Old state**	**New state**
CM_BUFFER_DATA	SEND	No change
	SEND_PENDING	SEND
CM_SEND_AND_FLUSH	SEND	No change
	SEND_PENDING	SEND
CM_SEND_AND_CONFIRM	SEND	No change
	SEND_PENDING	SEND
CM_SEND_AND_PREP_TO_ RECEIVE		RECEIVE
CM_SEND_AND_DEALLOCATE		RESET

For a *return_code* value of CM_PROGRAM_ERROR_PURGING or CM_SVC_ERROR_PURGING, the conversation changes to RECEIVE state. For other non-CM_OK values, the conversation changes to RESET state.

Remarks

The data collected in the local LU's send buffer is transmitted to the partner LU and partner program when one of the following occurs:

- The send buffer fills up.
- The local program issues a Flush, Confirm, or Deallocate call or other call that flushes the LU's send buffer. (Some send types, set by Set_Send_Type, include flush functionality.)

The data to be sent can be either:

- A complete data record on a mapped conversation. A complete data record is a string of the length specified by the *send_length* parameter.
- A complete logical record or portion thereof on a basic conversation. A complete logical record is determined by the LL value. (One logical record can end and a new one begin in the middle of the string of data to be sent.)

The LU does not automatically perform any conversion between ASCII and EBCDIC on the string of data to be sent. If necessary, the program can use the CSV CONVERT to translate a string from one character set to the other.

Send_Error

The **Send_Error** call (function name **cmserr**) notifies the partner program that the local program has encountered an application-level error.

```
CM_ENTRY Send_Error(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *request_to_send_received,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

request_to_send_received
Returned parameter. Specifies the request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program issued Request_To_Send, which requests the local program to change the conversation to RECEIVE state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program did not issue **Request_To_Send**. This value is not relevant if *return_code* is set to CM_PROGRAM_PARAMETER_CHECK or CM_STATE_CHECK.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

The value of *return_code* varies depending on the conversation state when the call is issued.

SEND State

If the program issues the call with the conversation in SEND state, the following return codes are possible:

CM_OK
Primary return code; the call executed successfully.

CM_OPERATION_NOT_ACCEPTED
Primary return code; a previous operation on this conversation is incomplete.

CM_OPERATION_INCOMPLETE
Primary return code; the operation has not completed (processing mode is nonblocking only) and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a Windows message and not call **Wait_For_Conversation**.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

CM_CONVERSATION_TYPE_MISMATCH
Primary return code; the partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

CM_PIP_NOT_SPECIFIED_CORRECTLY
Primary return code; the allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

CM_SECURITY_NOT_VALID
Primary return code; the user identifier or password specified in the allocation request was not accepted by the partner LU.

CM_SYNC LEVEL_NOT_SUPPORTED_PGM
Primary return code; the partner program does not support the synchronization level specified in the allocation request.

CM_TPN_NOT_RECOGNIZED
Primary return code; the partner LU does not recognize the program name specified in the allocation request.

CM_TP_NOT_AVAILABLE_NO_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

CM_TP_NOT_AVAILABLE_RETRY
Primary return code; the partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

CM_PROGRAM_ERROR_PURGING
Primary return code; one of the following occurred:

- While in RECEIVE or CONFIRM state, the partner program issued **Send_Error**. Data sent but not yet received is purged.
- While in SEND_PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

CM_RESOURCE_FAILURE_NO_RETRY
Primary return code; one of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

CM_RESOURCE_FAILURE_RETRY
Primary return code; the conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

CM_DEALLOCATED_ABEND
Primary return code; the conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND, or the remote LU has done so because of a remote program abnormal-ending condition. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.
- The remote TP terminated normally but did not deallocate the conversation before terminating. Node services at the remote LU deallocated the conversation on behalf of the remote TP.

CM_DEALLOCATED_ABEND_SVC
Primary return code; the conversation has been deallocated for one of the following reasons:

- The partner program issued **Deallocate** with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_DEALLOCATED_ABEND_TIMER
Primary return code; the conversation has been deallocated because the partner program issued **Deallocate** with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

CM_SVC_ERROR_PURGING
Primary return code; while in SEND state, the partner program or partner LU issued **Send_Error** with the type parameter set to SVC. Data sent to the partner program may have been purged.

RECEIVE State

If the call is issued in RECEIVE state, the following return codes are possible:

CM_OK
Primary return code; because incoming information is purged when the **Send_Error** call is issued in RECEIVE state, CM_OK is generated instead of the following:

CM_PROGRAM_ERROR_NO_TRUNC

CM_PROGRAM_ERROR_PURGING

CM_SVC_ERROR_NO_TRUNC

CM_SVC_ERROR_PURGING

CM_PROGRAM_ERROR_TRUNC

CM_SVC_ERROR_TRUNC (basic conversation only)

CM_PRODUCT_SPECIFIC_ERROR

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

For an explanation of these return codes, see Common Return Codes.

CM_DEALLOCATED_NORMAL
Primary return code; because incoming information is purged when **Send_Error** is issued in RECEIVE state, CM_DEALLOCATED_NORMAL is generated instead of the following:

CM_CONVERSATION_TYPE_MISMATCH

CM_PIP_NOT_SPECIFIED_CORRECTLY

CM_SECURITY_NOT_VALID

CM_SYNC_LEVEL_NOT_SUPPORTED_PGM

CM_TPN_NOT_RECOGNIZED

CM_TP_NOT_AVAILABLE_NO_RETRY

CM_TP_NOT_AVAILABLE_RETRY

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

SEND_PENDING State

If the call is issued in SEND_PENDING state, the following return codes are possible:

CM_OK (Primary return code; the call executed successfully.)

CM_PRODUCT_SPECIFIC_ERROR

CM_PROGRAM_ERROR_PURGING

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

CM_DEALLOCATED_ABEND

CM_DEALLOCATED_ABEND_SVC

CM_DEALLOCATED_ABEND_TIMER

CM_SVC_ERROR_PURGING

For an explanation of these return codes, see Common Return Codes.

CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE State

If the call is issued in CONFIRM, CONFIRM_SEND, or CONFIRM_DEALLOCATE state, the following return codes are possible:

CM_OK (Primary return code; the call executed successfully.)

CM_PRODUCT_SPECIFIC_ERROR

CM_RESOURCE_FAILURE_NO_RETRY

CM_RESOURCE_FAILURE_RETRY

For an explanation of these return codes, see Common Return Codes.

Other States

Issuing **Send_Error** with the conversation in RESET or INITIALIZE state is illegal. The following return codes are possible:

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid.
CM_PROGRAM_STATE_CHECK
Primary return code; the conversation state is not SEND, RECEIVE, CONFIRM, CONFIRM_SEND, CONFIRM_DEALLOCATE, or SEND_PENDING.

State Changes

The conversation can be in any state except INITIALIZE or RESET.

State changes, summarized in the following table, are based on the value of the *return_code* parameter.

return_code	**New state**
CM_OK	SEND
CM_CONVERSATION_TYPE_MISMATCH	RESET
CM_PIP_NOT_SPECIFIED_CORRECTLY	RESET
CM_SECURITY_NOT_VALID	RESET
CM_SYNC_LEVEL_NOT_SUPPORTED_PGM	RESET
CM_TPN_NOT_RECOGNIZED	RESET
CM_TP_NOT_AVAILABLE_NO_RETRY	RESET
CM_TP_NOT_AVAILABLE_RETRY	RESET
CM_RESOURCE_FAILURE_RETRY	RESET
CM_RESOURCE_FAILURE_NO_RETRY	RESET
CM_DEALLOCATED_ABEND	RESET
CM_DEALLOCATED_ABEND_PROG	RESET
CM_DEALLOCATED_ABEND_SVC	RESET
CM_DEALLOCATED_ABEND_TIMER	RESET
CM_DEALLOCATED_NORMAL	RESET
CM_PROGRAM_ERROR_PURGING	RECEIVE
CM_SVC_ERROR_PURGING	RECEIVE
All others	No change

Upon successful execution of this call, the conversation is in SEND state for the local program and in RECEIVE state for the partner program.

In a basic conversation, the local program can use Set_Log_Data to specify that error log data be sent to the partner LU and added to the local error log. If the conversation's log data length characteristic is greater than zero, the LU formats the data and stores it in the send buffer.

After **Send_Error** is completed, the log data length is set to zero and the log data to null.

If the conversation is in RECEIVE state when the program issues **Send_Error**, incoming data is purged by CPI-C. This data includes:

- Data sent by Send_Data.
- Confirmation requests.
- Deallocation requests if the conversation's deallocate type is set to CM_DEALLOCATE_CONFIRM or to CM_DEALLOCATE_SYNC_LEVEL with the synchronization level set to CM_CONFIRM.

CPI-C does not purge an incoming request-to-send indicator.

If the conversation is in SEND_PENDING state, the local program can issue Set_Error_Direction to specify whether the error being reported resulted from the received data or from the processing of the local program after successfully receiving the data.

Remarks

The local program can use **Send_Error** for such purposes as informing the partner program of an error encountered in received data, rejecting a confirmation request, or truncating an incomplete logical record it is sending.

Send_Error flushes the local LU's send buffer and sends the partner program the contents of the send buffer followed by the error notification.

The error notification is sent to the partner as one of the following *return_code* values:

- CM_PROGRAM_ERROR_TRUNC
- CM_PROGRAM_ERROR_NO_TRUNC
- CM_PROGRAM_ERROR_PURGING

Set_Conversation_Security_Password

The **Set_Conversation_Security_Password** call (function name **cmscsp**) is issued by the invoking program to specify the password required to gain access to the invoked program.

```
CM_ENTRY Set_Conversation_Security_Password(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *security_password,
  CM_INT32 FAR *security_password_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

security_password
Supplied parameter. Specifies the password required to gain access to the partner program. This parameter is a character string of up to eight ASCII characters and is case-sensitive. It must match the password for the user identifier configured for the partner program.

The allowed characters are:

- Uppercase and lowercase letters.
- Numerals 0 through 9.
- Special characters, except the space.

If the CPI-C automatic logon feature is to be used, this parameter must be set to the MS$SAME string. See the Remarks section for details.

security_password_length
Supplied parameter. Specifies the length of *security_password*. The range is from 0 through 8.

If the CPI-C automatic logon feature is to be used, this parameter must be set to 7. See the Remarks section for details.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *security_password_length* is out of range.

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The conversation is not in INITIALIZE state.
- The conversation's security type is not set to CM_SECURITY_PROGRAM.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

This call has an effect on the conversation only if the conversation security type is CM_SECURITY_PROGRAM or CM_SECURITY_SAME. It overrides the initial password from the side information specified by Initialize_Conversation. This call cannot be issued after Allocate has been issued.

An invalid password is not detected until the allocation request, generated by **Allocate**, is sent to the partner LU. The error is returned to the invoking program on a subsequent call.

Automatic logon for CPI-C applications is supported by Microsoft® Host Integration Server, Microsoft® SNA Server 4.0, and Microsoft® SNA Server 3.0 with Service Pack 1 or later. This feature requires specific configuration by the network administrator: The CPI-C application must be invoked on the LAN side from a client of SNA server. The client must be logged into a Microsoft® Windows 2000 or Microsoft® Windows NT® domain, but can be any platform that supports Host Integration Server or SNA server CPI-C APIs.

The client application is coded to use "program" level security, with a special hard-coded CPI-C user name MS$SAME and password MS$SAME. When this session allocation flows from client to SNA server, the SNA server looks up the host account and password corresponding to the Windows 2000 or Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

Set_Conversation_Security_Type

The **Set_Conversation_Security_Type** call (function name **cmscst**) is issued by the invoking program to specify the information the partner LU requires to validate access to the invoked program.

```
CM_ENTRY Set_Conversation_Security_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_security_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

conversation_security_type
Supplied parameter. Specifies the information the partner LU requires to validate access to the invoked program. Based on the conversation security established for the invoked program during configuration, use one of the following values:

CM_SECURITY_NONE
To indicate that the invoked program uses no conversation security.

CM_SECURITY_PROGRAM
To indicate that the invoked program uses conversation security and thus requires a user identifier and password.

CM_SECURITY_SAME
To indicate that the user ID is sent on the allocate request to node services in the partner LU. This setting is also used to specify that the invoked program, invoked with a valid user identifier and password, in turn invokes another program (as illustrated in Communication Between TPs). For example, assume that program A invokes program B with a valid user identifier and password, and program B in turn invokes program C. If program B specifies the value CM_SECURITY_SAME, CPI-C will send the LU for program C, the user identifier from program A, and an already-verified indicator. This indicator tells program C not to require the password (if program C is configured to accept an already-verified indicator).

When CM_SECURITY_SAME is used, your application must always call Set_Conversation_Security_User_ID and Set_Conversation_Security_Password to provide values for the *security_user_ID* and *security_password* parameters. Depending on the properties negotiated between SNA server and the peer LU, the **Allocate** function will send one of 3 kinds of Attach (FMH-5) messages, in this order of precedence:

1. If the LUs have negotiated "already verified" security, then the Attach sent by SNA server will not include the contents of the security_password parameter field specified by Set_Conversation_Security_Password.
2. If the LUs have negotiated "persistent verification" security, then the Attach sent by SNA server will include the security_password parameter specified by Set_Conversation_Security_Password, but only when the Attach is the first for the specified security_user_ID parameter set by Set_Conversation_Security_User_ID since the start of the LU-LU session, and will omit the security_password parameter on all subsequent Attaches (issued by your application or any other application using this LU-LU-mode triplet).
3. Your application cannot tell which mode of security has been negotiated between the LUs, nor can it tell whether the Allocate function it is issuing is the first for that LU-LU-mode triplet. So your application must always call Set_Conversation_Security_User_ID and Set_Conversation_Security_Password to set the security_user_ID and security_password parameters when conversation_security_type is set to CM_SECURITY_SAME.

For more information on persistent verification and already verified security, see the SNA Formats Guide, section "FM Header 5: Attach (LU 6.2)."

If the CPI-C automatic logon feature is to be used, this parameter must be set to CM_SECURITY_PROGRAM. See the Remarks section for details.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* or *conversation_security_type* is invalid.
CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

This call overrides the initial security type from the side information specified by Initialize_Conversation. This call cannot be issued after Allocate has been issued.

If the conversation security type is set to CM_SECURITY_NONE, the user identifier and password are ignored when the conversation is allocated.

A conversation security type of CM_SECURITY_SAME is intended for use between nodes which have the same set of user IDs and which accept user validation performed on one node as validating the user for all nodes. A password is not used in this case except for the initial validation of the user ID.

Automatic logon for CPI-C applications is supported by Microsoft® Host Integration Server, Microsoft® SNA Server 4.0, and Microsoft® SNA Server 3.0 with Service Pack 1 or later. This feature requires specific configuration by the network administrator: The CPI-C application must be invoked on the LAN side from a client of SNA server. The client must be logged into a Microsoft® Windows 2000 or Microsoft® Windows NT® domain, but can be any platform that supports SNA server CPI-C APIs.

The client application is coded to use "program" level security, with a special hard-coded CPI-C user name MS$SAME and password MS$SAME. When this session allocation flows from client to SNA server, the SNA server looks up the host account and password corresponding to the Windows 2000 or Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

Set_Conversation_Security_User_ID

The **Set_Conversation_Security_User_ID** call (function name **cmscsu**) is issued by the invoking program to specify the user identifier required to gain access to the invoked program.

```
CM_ENTRY Set_Conversation_Security_User_ID(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *security_user_ID,
  CM_INT32 FAR *security_user_ID_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

security_user_ID
Supplied parameter. Specifies the user identifier required to gain access to the partner program. This parameter is a character string of up to eight ASCII characters and is case-sensitive.

The allowed characters are:

- Uppercase and lowercase letters.
- Numerals 0 through 9.
- Special characters, except the space.

If the CPI-C automatic logon feature is to be used, this parameter must be set to the MS$SAME string. See the Remarks section for details.

security_user_ID_length
Supplied parameter. Specifies the length of *security_user_ID*. The range is from 0 through 8.

If the CPI-C automatic logon feature is to be used, this parameter must be set to 7. See the Remarks section for details.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *security_user_ID_length* is out of range.

CM_PROGRAM_STATE_CHECK
Primary return code; one of the following occurred:

- The conversation is not in INITIALIZE state.
- The conversation's security type is not set to CM_SECURITY_PROGRAM.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

This call has an effect on the conversation only if the conversation security type is CM_SECURITY_PROGRAM or

CM_SECURITY_SAME. It overrides the initial user identifier from the side information specified by Initialize_Conversation. This call cannot be issued after Allocate has been issued.

An invalid user identifier is not detected until the allocation request, generated by **Allocate**, is sent to the partner LU. The error is returned to the invoking program on a subsequent call.

Automatic logon for CPI-C applications is supported by Microsoft® Host Integration Server, Microsoft® SNA Server 4.0, and Microsoft® SNA Server 3.0 with Service Pack 1 or later and by. This feature requires specific configuration by the network administrator: The CPI-C application must be invoked on the LAN side from a client of SNA server. The client must be logged into a Microsoft® Windows 2000 or Microsoft® Windows NT® domain, but can be any platform that supports SNA server CPI-C APIs.

The client application is coded to use "program" level security, with a special hard-coded CPI-C user name MS$SAME and password MS$SAME. When this session allocation flows from client to SNA server, the SNA server looks up the host account and password corresponding to the Windows 2000 or Windows NT account under which the client is logged in, and substitutes the host account information into the APPC attach message it sends to the host.

Set_Conversation_Type

The **Set_Conversation_Type** call (function name **cmsct**) is issued by the invoking program to define a conversation as being mapped or basic. This call overrides the default conversation type established by Initialize_Conversation. The default conversation type is CM_MAPPED_CONVERSATION. This call cannot be issued after Allocate has been issued.

```
CM_ENTRY Set_Conversation_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

conversation_type
Supplied parameter. Specifies the type of conversation to be allocated by **Allocate**. Possible values are:

CM_BASIC_CONVERSATION

CM_MAPPED_CONVERSATION

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* or *conversation_type* is invalid.
- The *conversation_type* parameter specifies a mapped conversation, but the fill characteristic is set to CM_FILL_BUFFER, which is incompatible with mapped conversations. Before changing the conversation type to mapped, you must issue the Set_Fill call to change the fill type to CM_FILL_LL.
- The *conversation_type* parameter specifies a mapped conversation. However, a previous Set_Log_Data call, allowed only in basic conversations, is still in effect.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Set_CPIC_Side_Information

The **Set_CPIC_Side_Information** call (function name **xcmssi**) adds or replaces a side information entry in memory. A CPI-C side information entry associates a set of conversation characteristics with a symbolic definition name. This call overrides entries having the same symbolic destination name.

```
CM_ENTRY Set_CPIC_Side_Information(
  unsigned char FAR *key_lock,
  SIDE_INFO FAR *side_info_entry,
  CM_INT32 FAR *side_info_entry_length,
  CM_INT32 FAR *return_code
);
```

Parameters

key_lock
Supplied parameter. This parameter is ignored.

side_info_entry
Supplied parameter. Specifies the contents of a side information entry. The following table describes the *side_info_entry* structure, which defines the format of the side information entry:

Offset	Description	Type	Length
0	*sym_dest_name*	unsigned char	8 bytes
8	*partner_LU_name*	unsigned char	17 bytes
25	*reserved*	unsigned char	3 bytes
28	*TP_name_type*	signed long int	32 bits
32	*TP_name*	unsigned char	64 bytes
96	*mode_name*	unsigned char	8 bytes
104	*conversation_security_type*	signed long int	32 bits
108	*security_user_ID*	unsigned char	8 bytes
116	*security_password*	unsigned char	8 bytes

The allowed characters for *sym_dest_name* are the uppercase letters (A through Z) and the numerals 0 through 9.

Set_CPIC_Side_Information is the only CPI-C call that lets you specify an SNA service TP as the partner program. The SNA convention for naming a service TP is up to four characters. The first character is a hexadecimal byte between 0x00 and 0x3F. The remaining characters are translated from ASCII to EBCDIC.

For the allowed characters for the other fields, see the description of the corresponding **Set_** call. For example, for the *mode_name* field, see the description of the Set_Mode_Name call.

Each field in the structure must be left-aligned. Pad fields on the right with spaces as necessary.

side_info_entry_length
Supplied parameter. Specifies the length of *side_info_entry*. It is always 124.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- A value specified in the *side_info_entry* structure is invalid.
- The left character of the *side_info_entry* contains a space.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state.

There is no state change.

Remarks

Invalid string parameters in the side information (for example, specifying a nonexistent partner LU) are not detected until Allocate is issued. The error is returned on a call following **Allocate**.

Set_Deallocate_Type

The **Set_Deallocate_Type** call (function name **cmsdt**) specifies how the conversation is to be deallocated.

```
CM_ENTRY Set_Deallocate_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *deallocate_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

deallocate_type
Supplied parameter. Specifies how to perform the deallocation. Possible values are:

CM_DEALLOCATE_ABEND
Indicates that the conversation is to be deallocated abnormally and unconditionally. A program should specify CM_DEALLOCATE_ABEND when it encounters an error preventing the successful completion of a transaction.

If the conversation is in SEND state, CPI-C sends the contents of the local LU's send buffer to the partner program before deallocating the conversation. If the conversation is in RECEIVE state, incoming data can be purged. For a basic conversation in SEND state, logical record truncation can occur.

CM_DEALLOCATE_CONFIRM
Is used to send the partner program the contents of the local LU's send buffer and a request to confirm the deallocation.

This request for deallocation confirmation is sent by Deallocate or by Send_Data with the send type set to CM_SEND_AND_DEALLOCATE. The conversation is deallocated normally when the partner program issues Confirmed, responding to the confirmation request.

CM_DEALLOCATE_FLUSH
Is used to send the contents of the local LU's send buffer to the partner program before deallocating the conversation normally.

CM_DEALLOCATE_SYNC_LEVEL
Uses the conversation's synchronization level to determine how to deallocate the conversation. A default synchronization level is established by Initialize_Conversation and can be overridden by Set_Sync_Level.

If the synchronization level of the conversation is CM_NONE, the default, the contents of the local LU's send buffer are sent to the partner program and the conversation is deallocated normally.

If the synchronization level of the conversation is CM_CONFIRM, the contents of the local LU's send buffer and a request to confirm the deallocation are sent to the partner program. This request for deallocation confirmation is sent by Deallocate or by Send_Data with the send type set to CM_SEND_AND_DEALLOCATE. The conversation is deallocated normally when the partner program issues the Confirmed call, responding to the confirmation request.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* or *deallocate_type* is invalid.
- The *deallocate_type* parameter specifies CM_DEALLOCATE_CONFIRM, but the conversation's synchronization level is set to CM_NONE.
- The address of a variable is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

This call overrides the default deallocate type established by Initialize_Conversation or Accept_Conversation. The default deallocate type is CM_DEALLOCATE_SYNC_LEVEL.

The deallocation instructions specified by this call take effect when Deallocate is issued or when the send type is set to CM_SEND_AND_DEALLOCATE and Send_Data is issued.

You can set *deallocate_type* to CM_FLUSH if the synchronization level of the conversation is set to CM_NONE or CM_CONFIRM.

The value CM_DEALLOCATE_FLUSH is functionally the same as CM_DEALLOCATE_SYNC_LEVEL with the conversation's synchronization level set to CM_NONE.

The value CM_DEALLOCATE_CONFIRM is functionally the same as CM_DEALLOCATE_SYNC_LEVEL with the conversation's synchronization level set to CM_CONFIRM.

Set_Error_Direction

The **Set_Error_Direction** call (function name **cmsed**) specifies whether a program detected an error while receiving data or while preparing to send data.

```
CM_ENTRY Set_Error_Direction(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *error_direction,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

error_direction
Supplied parameter. Specifies the direction in which data was flowing when the program encountered an error. Possible values are:

CM_RECEIVE_ERROR
An error occurred in the data received from the partner program.

CM_SEND_ERROR
An error occurred while the local program prepared to send data to the partner program.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* or *error_direction* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

This call overrides the default error direction established by Initialize_Conversation or Accept_Conversation. The default error direction is CM_RECEIVE_ERROR.

Error direction is relevant only when a program issues Send_Error in SEND_PENDING state, immediately after issuing Receive and receiving data (*data_received* is a value other than CM_NO_DATA_RECEIVED) and a send indicator (*status_received* is CM_SEND_RECEIVED).

When the conversation is in SEND_PENDING state, the program issues **Send_Error** if it detects errors in the received data or if an error occurred while the local program prepared to send data. The program must supply the error direction information using **Set_Error_Direction** before issuing **Send_Error** because the LU cannot tell which kind of error occurred (receive or send). The new error direction remains in effect until a subsequent **Set_Error_Direction** changes it.

When **Send_Error** is issued, the partner program receives one of the following return codes:

- CM_PROGRAM_ERROR_PURGING if *error_direction* is set to CM_RECEIVE_ERROR
- CM_PROGRAM_ERROR_NO_TRUNC if *error_direction* is set to CM_SEND_ERROR

Set_Fill

The **Set_Fill** call (function name **cmsf**) specifies whether programs will receive data in the form of logical records or as a specified length of data. This call is allowed only in basic conversations.

```
CM_ENTRY Set_Fill(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *fill,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

fill
Supplied parameter. Specifies the form in which programs will receive data. The following are possible choices:

CM_FILL_BUFFER
The local program receives data until the number of bytes specified by the *requested_length* parameter of the **Receive** call is reached or until the end of the data. Data is received without regard for the logical-record format.

CM_FILL_LL
Data is received in logical-record format. The data received can be a complete logical record, a portion of a logical record equal to the *requested_length* parameter of the Receive call, or the end of a logical record.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* or *fill* is invalid.
- The current conversation is mapped.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

Set_Fill overrides the default *fill* established by Initialize_Conversation or Accept_Conversation. The default *fill* is CM_FILL_LL.

The *fill* value affects all subsequent Receive calls. It can be changed by reissuing the **Set_Fill** call.

Set_Log_Data

The **Set_Log_Data** call (function name **cmsld**) specifies a log message (log data) and its length to be sent to the partner LU. This call is allowed only in basic conversations. It overrides the default log data, which is null, and the default log data length, which is zero.

```
CM_ENTRY Set_Log_Data(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *log_data,
  CM_INT32 FAR *log_data_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

log_data
Supplied parameter. Specifies the starting address of the data to be sent to the partner LU. It can contain up to 512 ASCII characters. The allowed characters are:

- Uppercase and lowercase letters.
- Numerals 0 through 9.
- Special characters.
- The space.

log_data_length
Supplied parameter. Specifies the length of the log data. The range is from 0 through 512 bytes.

A length of 0 indicates that there is no log data, and the *log_data* parameter is ignored.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The conversation type is set to mapped.
- The value specified by *log_data_length* is out of range (greater than 512 or less than 0).

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

The log data specified by **Set_Log_Data** is sent to the partner LU when the local program issues one of the following calls:

- Send_Error
- Deallocate with the conversation's deallocate type set to CM_DEALLOCATE_ABEND
- Send_Data with the conversation's send type set to CM_SEND_AND_DEALLOCATE and the deallocate type set to CM_DEALLOCATE_ABEND

After sending the log data to the partner LU, the local LU resets the log data to null and the log data length to zero.

CPI-C automatically converts the log data from ASCII to other encoding standards, such as EBCDIC, as required.

Set_Mode_Name

The **Set_Mode_Name** call (function name **cmsmn**) is issued by the invoking program to specify the mode name for a conversation. This call overrides the system-defined mode name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after Allocate has been issued. Issuing this call has no effect on the side information itself.

```
CM_ENTRY Set_Mode_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *mode_name,
  CM_INT32 FAR *mode_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

mode_name
Supplied parameter. Specifies the starting address of the mode name (the name of a set of networking characteristics defined during configuration). The mode name can contain up to eight ASCII characters. The allowed characters are:

- Uppercase letters.
- Numerals 0 through 9.

The value of *mode_name* must match the name of a mode associated with the partner LU during configuration. The mode name cannot be SNASVCMG or CPSVCMG.

mode_name_length
Supplied parameter. Specifies the length of the mode name. The range is from 0 through 8 bytes

If *mode_name_length* is set to zero, **Set_Mode_Name** is ignored.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *mode_name_length* is out of range (greater than 8 or less than 0).

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

Specifying an invalid value for *mode_name* is not detected until Allocate is issued.

Set_Partner_LU_Name

The **Set_Partner_LU_Name** call (function name **cmspln**) is issued by the invoking program to specify the partner LU name. This call overrides the partner LU name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after Allocate has been issued. Issuing this call has no effect on the side information itself.

```
CM_ENTRY Set_Partner_LU_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *partner_LU_name,
  CM_INT32 FAR *partner_LU_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

partner_LU_name
Supplied parameter. Specifies the starting address of the partner LU name. The mode name can contain up to 17 ASCII characters. The allowed characters are:

- Uppercase letters.
- Numerals 0 through 9.

The partner LU name can be either:

- An alias consisting of one through eight characters.
- A fully qualified network name consisting of from 2 through 17 characters. A period separates the network identifier (which can be from zero through eight characters) from the network LU name (which can be from one through eight characters). If the network identifier is zero characters long, the period is still required.

The partner LU name must match the name of a partner LU established during configuration.

partner_LU_name_length
Supplied parameter. Specifies the length of the partner LU name. The range is from 1 through 17.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *partner_LU_name_length* is out of range (greater than 17 or less than 1).

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

Specifying an invalid value for *partner_LU_name* is not detected until Allocate is issued.

Set_Prepare_To_Receive_Type

The **Set_Prepare_To_Receive_Type** call (function name **cmsptr**) specifies how the subsequent Prepare_To_Receive calls will be executed. It overrides the default prepare-to-receive processing established by Initialize_Conversation or Accept_Conversation. By default, the prepare-to-receive processing is based on the synchronization level of the conversation.

The prepare-to-receive type affects all subsequent **Prepare_To_Receive** calls. It can be changed by reissuing **Set_Prepare_To_Receive_Type**.

```
CM_ENTRY Set_Prepare_To_Receive_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *prepare_to_receive_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

prepare_to_receive_type
Supplied parameter. Specifies how subsequent Prepare_To_Receive calls will be executed. Possible values are:

CM_PREP_TO_RECEIVE_CONFIRM
Is used to send the partner program the contents of the LU's send buffer and a confirmation request. Upon receipt of confirmation, the conversation changes to RECEIVE state.

CM_PREP_TO_RECEIVE_FLUSH
Is used to send the partner program the contents of the local LU's send buffer and changes the conversation to RECEIVE state.

CM_PREP_TO_RECEIVE_SYNC_LEVEL
Is used by the conversation's synchronization level to determine prepare-to-receive processing. A default synchronization level is established by Initialize_Conversation and can be overridden by Set_Sync_Level.

If the synchronization level of the conversation is CM_NONE, the default, the contents of the local LU's send buffer are sent to the partner program and the conversation changes to RECEIVE state. If the synchronization level of the conversation is CM_CONFIRM, the contents of the local LU's send buffer and a request for confirmation are sent to the partner program. The conversation changes to RECEIVE state when the partner program issues Confirmed, responding to the confirmation request.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *prepare_to_receive_type* or *conversation_ID* is invalid.
- The *prepare_to_receive_type* parameter is set to CM_PREP_TO_RECEIVE_CONFIRM, but the conversation's synchronization level is set to CM_NONE.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Set_Processing_Mode

The **Set_Processing_Mode** call (function name **cmspm**) specifies for the conversation whether subsequent calls will be returned when the operation they have requested is complete (blocking) or immediately after the operation is initiated (nonblocking).

> **Note** A program is notified of the completion of nonblocking calls when it issues Wait_For_Conversation or through a Windows message sent to a WndProc identified by the hWnd in the Specify_Windows_Handle call.

```
CM_ENTRY Set_Processing_Mode(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *receive_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

receive_type
Supplied parameter. Specifies whether subsequent calls on the conversation will be blocking or nonblocking. Possible values are:

CM_BLOCKING
Subsequent calls will return only when the operation is complete.

CM_NON_BLOCKING
Subsequent calls will return immediately once the operation has been initiated.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the previous incomplete operation on the conversation has not yet completed.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* or *processing_mode* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Set_Receive_Type

The **Set_Receive_Type** call (function name **cmsrt**) specifies how the program will receive data on subsequent Receive calls. It overrides the default receive type established by the Initialize_Conversation or Accept_Conversation call. By default, the program waits for data to arrive if it is not available when the **Receive** call is issued.

The receive type value affects all subsequent **Receive** calls. It can be changed by reissuing **Set_Receive_Type**.

```
CM_ENTRY Set_Receive_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *receive_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

receive_type
Supplied parameter. Specifies how data is to be received by the program on the subsequent Receive calls. Possible values are:

CM_RECEIVE_AND_WAIT
The local program receives any data that is currently available from the partner program. If no data is currently available, the local program waits for data to arrive.

CM_RECEIVE_IMMEDIATE
The local program receives any data currently available from the partner program. If no data is available, the local program does not wait.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* or *receive_type* is invalid, or the address of a variable is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Set_Return_Control

The **Set_Return_Control** call (function name **cmsrc**) is issued by the invoking program to specify when the local LU, acting on the session request from the local program's Allocate call, should return control to the local program.

```
CM_ENTRY Set_Return_Control(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *return_control,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

return_control
Supplied parameter. Specifies when the local LU, acting on the Allocate call, should return control to the local program. The following are allowed values:

CM_IMMEDIATE
The LU allocates a contention-winner session, if one is immediately available, and returns control to the program.

CM_WHEN_SESSION_ALLOCATED
The LU does not return control to the program until it allocates a session or encounters errors. If a session is not available, the program waits for one. (If the session limit is zero, the LU returns control immediately.)

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* or *return_control* is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Remarks

This call overrides the default return control established by Initialize_Conversation. By default, control is returned when the session is allocated. This call cannot be issued after the Allocate call has been issued.

For further information about sessions, see Writing CPI-C Applications.

If the LU is unable to allocate a session, the notification is returned on the **Allocate** call.

Set_Send_Type

The **Set_Send_Type** call (function name **cmsst**) specifies how data will be sent by the next Send_Data call. It overrides the default send type established by Initialize_Conversation or Accept_Conversation. The default send type is CM_BUFFER_DATA, indicating that data only (and no control information) is to be sent.

The *send_type* value affects all subsequent **Send_Data** calls. It can be changed by reissuing **Set_Send_Type**.

```
CM_ENTRY Set_Send_Type(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *send_type,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

send_type
Supplied parameter. Specifies how data is sent by the next Send_Data call. Possible values are:

CM_BUFFER_DATA
The data pointed to by **Send_Data** is stored in a buffer until the buffer fills up or is flushed.

CM_SEND_AND_FLUSH
The data pointed to by **Send_Data** is to be sent immediately.

CM_SEND_AND_CONFIRM
The data is to be sent immediately with a request for confirmation.

CM_SEND_AND_PREP_TO_RECEIVE
The data is to be sent immediately along with notification to the partner program that the conversation state for the sending program is changing to RECEIVE.

CM_SEND_AND_DEALLOCATE
The data is to be sent immediately along with deallocation notification.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* or *send_type* is invalid.
- The *send_type* parameter is set to CM_SEND_AND_CONFIRM, but the conversation's synchronization level is set to CM_NONE.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation can be in any state except RESET.

There is no state change.

Remarks

The *send_type* values that cause additional information to be sent with the data pointed to by Send_Data let you economize on the number of calls issued. The following table summarizes **Send_Data** equivalences.

Send_Data with *send_type* set to this value	**Equates to Send_Data with *send_type* set to CM_BUFFER_DATA followed by**
CM_SEND_AND_FLUSH	Flush
CM_SEND_AND_CONFIRM	Confirm
CM_SEND_AND_PREP_TO_RECEIVE	Prepare_To_Receive

CM_SEND_AND_DEALLOCATE	Deallocate

Set_Sync_Level

The **Set_Sync_Level** call (function name **cmssl**) is issued by the invoking program to specify the synchronization level of the conversation. The synchronization level determines whether the programs synchronize their processing through the Confirm and Confirmed calls.

This call overrides the synchronization level established by the Initialize_Conversation call. The default synchronization level is CM_NONE, indicating no synchronization. This call cannot be issued after the Allocate call has been issued.

```
CM_ENTRY Set_Sync_Level(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *sync_level,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by **Initialize_Conversation**.

sync_level
Supplied parameter. Specifies the synchronization level of the conversation. Possible values are:

CM_NONE
The programs will not perform confirmation processing.

CM_CONFIRM
The programs can perform confirmation processing.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* or *sync_level* is invalid.
- The *sync_level* parameter specifies CM_NONE but one of the following has occurred: the *send_type* parameter is set to CM_SEND_AND_CONFIRM, or the *prepare_to_receive_type* parameter is set to CM_PREP_TO_RECEIVE_CONFIRM, or the *deallocate_type* is set to CM_DEALLOCATE_CONFIRM.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Set_TP_Name

The **Set_TP_Name** call (function name **cmstpn**) is issued by the invoking program to specify the partner (invokable) program name. This call overrides the partner program name derived from the side information when the Initialize_Conversation call was issued. This call cannot be issued after the Allocate call has been issued. Issuing this call has no effect on the side information itself.

```
CM_ENTRY Set_TP_Name(
  unsigned char FAR *conversation_ID,
  unsigned char FAR *TP_name,
  CM_INT32 FAR *TP_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation.

TP_name
Supplied parameter. Specifies the starting address of the partner program name. The program name can contain up to 64 ASCII characters. The allowed characters are:

- Uppercase and lowercase letters.
- Numerals 0 through 9.
- Special characters, except the space.

You cannot use **Set_TP_Name** to specify the name of an SNA service TP. You can, however, use Set_CPIC_Side_Information to do this.

Double-byte character sets, such as Kanji, are not supported.

TP_name_length
Supplied parameter. Specifies the length of the partner program name. The range is from 1 through 64.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is not in INITIALIZE state.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The value specified by *conversation_ID* is invalid.
- The value specified by *TP_name_length* is out of range (greater than 64 or less than 1).
- The address of a variable is invalid.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in INITIALIZE state.

There is no state change.

Specify_Local_TP_Name

The **Specify_Local_TP_Name** call (function name **cmsltp**) is issued by the program to indicate that it is able to accept incoming conversations that are directed to the name given.

```
CM_ENTRY Specify_Local_TP_Name(
  unsigned char FAR *TP_name,
  CM_INT32 FAR *TP_name_length,
  CM_INT32 FAR *return_code
);
```

Parameters

TP_name
Supplied parameter. Specifies the starting address of the local TP name. The program name can contain up to 64 ASCII characters. The allowed characters are:

- Uppercase and lowercase letters.
- Numerals 0 through 9.
- Special characters, except the space.

You cannot use **Specify_Local_TP_Name** to specify the name of an SNA service TP.

Double-byte character sets, such as Kanji, are not supported.

TP_name_length
Supplied parameter. Specifies the length of the local program name. The range is from 1 through 64.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; one of the following occurred:

- The *TP_name* supplied is invalid.
- The value specified by *TP_name_length* is out of range (greater than 64 or less than 1).

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The call is not associated with a particular conversation, and so no state restrictions apply.

There is no state change.

Remarks

A program can issue this call more than once to handle incoming conversations with more than one TP name. The program can discover the actual name on the incoming conversation by calling Extract_TP_Name.

Specify_Windows_Handle

The **Specify_Windows_Handle** call (function name **xchwnd**) sets the Windows handle to which a message is sent on completion of an operation in nonblocking mode.

```
CM_ENTRY Specify_Windows_Handle(
  HWND hwndNotify,
  CM_INT32 FAR *return_code
);
```

Parameters

hwndNotify
Supplied parameter. Specifies the Windows handle to be notified when the outstanding operation completes.
return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.
CM_PROGRAM_PARAMETER_CHECK
The Windows handle is invalid.
CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The state change is dependent on the operation that completed and its return code.

Remarks

An application can set the processing mode by calling Set_Processing_Mode. If the Windows handle is set to NULL, or this call is never issued, then the application must call Wait_For_Conversation to be notified when the outstanding operation completes.

When an asynchronous operation is complete, the application's window *hwndNotify* receives the message returned by **RegisterWindowMessage** with "WinAsyncCPIC" as the input string. The *wParam* value contains the *conversation_return_code* from the operation that is completing. Its values will depend on which operation was originally issued. The *lParam* argument contains the CM_PTR to the *conversation_ID* specified in the original function call.

Test_Request_To_Send_Received

The **Test_Request_To_Send_Received** call (function name **cmtrts**) determines whether a request-to-send notification has been received from the partner program.

```
CM_ENTRY Test_Request_To_Send_Received(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *request_to_send_received,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Supplied parameter. Specifies the identifier for the conversation. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

request_to_send_received
Returned parameter. The request-to-send-received indicator. Possible values are:

CM_REQ_TO_SEND_RECEIVED
The partner program issued Request_To_Send, which requests the local program to change the conversation to RECEIVE state.

CM_REQ_TO_SEND_NOT_RECEIVED
The partner program did not issue **Request_To_Send**. This value is not relevant if *return_code* contains a value other than CM_OK.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_PROGRAM_PARAMETER_CHECK
Primary return code; the value specified by *conversation_ID* is invalid, or the address of a variable is invalid.

CM_PROGRAM_STATE_CHECK
Primary return code; the conversation is in a state other than SEND, RECEIVE, or SEND_PENDING.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error occurred and has been logged in the product's error log.

State Changes

The conversation must be in SEND, RECEIVE, or SEND_PENDING state.

There is no state change.

Wait_For_Conversation

The **Wait_For_Conversation** call (function name **cmwait**) waits for an operation to complete that has been initiated when the *processing_mode* conversation characteristic was set to CM_NON_BLOCKING and CM_OPERATION_INCOMPLETE was returned in the *return_code* parameter.

```
CM_ENTRY Wait_For_Conversation(
  unsigned char FAR *conversation_ID,
  CM_INT32 FAR *conversation_return_code,
  CM_INT32 FAR *return_code
);
```

Parameters

conversation_ID
Returned parameter. Specifies the identifier for the conversation on which the operation completed. The value of this parameter was returned by Initialize_Conversation or Accept_Conversation.

conversation_return_code
Returned parameter. Specifies the *return_code* from the operation that is completing. Its values will depend on which operation was originally issued.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
Primary return code; the call executed successfully.

CM_SYSTEM_EVENT
Primary return code; the wait completed not because the operation completed but because some system event occurred.

CM_PROGRAM_STATE_CHECK
Primary return code; the program has no incomplete operation outstanding.

CM_PRODUCT_SPECIFIC_ERROR
Primary return code; a product-specific error has occurred and has been logged in the product's error log.

State Changes

The state change is dependent on the operation that completed and its return code.

Remarks

The program must have an incomplete operation outstanding on some conversation.

See Also

Set_Processing_Mode, Specify_Windows_Handle

CPI-C Functions Not Supported

The Windows CPI-C implementation does not support the following CPI-C 1.2 functions:

CPI-C Function	Function Name
Extract_Conversation_Context	cmectx
Extract_Maximum_Buffer_Size	cmembs
Extract_Secondary_Information	cmesi
Extract_Send_Receive_Mode	cmesrm
Extract_TP_ID	xceti
Initialize_Conversation_For_TP	xcinct
Initialize_For_Incoming	cminic
Receive_Expedited	cmrcvx
Release_Local_TP_Name	cmrltp
Send_Expedited	cmsndx
Set_Queue_Callback_Function	cmsqcf
Set_Queue_Processing_Mode	cmsqpm
Set_Send_Receive_Mode	cmssrm
Start_TP	xcstp
Wait_For_Completion	cmwcmp
End_TP	xcendt

Extensions for the Windows Environment

This section describes API extensions to Windows CPI-C that allow nonblocking or asynchronous verb completion. Asynchronous verbs return control to the program immediately, without waiting for full execution, and must notify the application later when the verb has been completed. An application is also notified in response to the completion of a Wait_For_Conversation call. In contrast, synchronous verbs block—the function call does not return until the call has completed.

Under Microsoft® Windows® 2000, Microsoft® Windows NT®, Microsoft Windows 98, and Microsoft Windows 95, two methods are available for handling asynchronous verb completion:

- Message posting using window handles.
- Waiting on Win32® events.

The first method uses messages posted to a window handle to notify an application of verb completion. This method using window handles and messages is also supported on Windows 3.x. There is one such window for each CPI-C application. Each CPI-C conversation can have one asynchronous verb outstanding at any time. When a verb completes, the posting to the window takes as parameters the CPI-C conversation identifier of the verb that has completed, and the return code of the verb.

The extensions using window handles and message posting described in this section have been designed for all implementations and versions of Microsoft Windows from version 3.0 through the latest versions of Windows 2000, Windows NT, Windows 98, and Windows 95. They provide compatibility for Windows programming and optimum application performance in the 16-bit operating environment.

A second method using Win32 events for notification is supported on Microsoft® Host Integration Server and Microsoft® SNA Server version 3.0 and later. The extensions using Win32 events described in this section (WinCPICSetEvent and WinCPICExtractEvent) operate only on Windows 2000, Windows NT, Windows 98, and Windows 95, and offer the optimum application performance in the 32-bit operating environment. If an event has been registered with the conversation, then an application can call the Win32 **WaitForSingleObject** or **WaitForMultipleObjects** function to wait to be notified of the completion of the verb.

Windows CPI-C allows multithreaded Windows-based processes. Multithreading is the running of several processes in rapid sequence within a single program. A process contains one or more threads of execution. The 16-bit Windows environment is not multithreaded. In this instance, a task corresponds to a process with a single thread. All references to threads in this document refer to actual threads in multithreaded Windows environments. Using Win32 events is a common technique used by a multithreaded application that can dedicate a thread or several threads to handle event handling.

The extension descriptions in this section provide a definition of the function, syntax, return values, and remarks for using these Windows extensions in CPI-C programs.

WinCPICCleanup

The **WinCPICCleanup** function terminates and deregisters an application from a Windows CPI-C implementation.

```
BOOL WINAPI WinCPICCleanup(void);
```

Return Values

The return value specifies whether the deregistration was successful. If the value is not zero, the application was successfully deregistered. The application was not deregistered if a value of zero is returned.

Remarks

Use **WinCPICCleanup** to indicate deregistration of a Windows CPI-C application from a Windows CPI-C implementation.

WinCPICExtractEvent

The **WinCPICExtractEvent** function provides a method for an application to determine the event handle being used for a CPI-C conversation.

```
VOID WINAPI WinCPICExtractEvent(
    unsigned char FAR *conversation_ID,
    HANDLE FAR *event_handle,
    CM_INT32 FAR *return_code
   );
```

Parameters

conversation_ID
Specifies the identifier for the conversation for which this event is used. This parameter is returned by the initial Accept_Conversation call.

event_handle
Returned parameter. The handle of the event being used by this conversation. If no handle has been registered, this parameter returns as a NULL.

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
The function executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One or more of the parameters passed to this function are invalid.

Remarks

When a verb is issued on a nonblocking conversation, it returns CM_OPERATION_INCOMPLETE if it is going to complete asynchronously. If an event has been registered with the conversation, then the application can call **WaitForSingleObject** or **WaitForMultipleObjects** to be notified of the completion of the verb. **WinCPICExtractEvent** allows a CPI-C application to determine this event handle. When the verb has completed, the application must call Wait_for_Conversation to determine the return code for the asynchronous verb. The Cancel_Conversation function can be called to cancel an operation and conversation.

If no event has been registered, then the asynchronous verb completes as it does at present, which is by posting a message to the window that the application has registered with the CPI-C library.

WinCPICIsBlocking

The **WinCPICIsBlocking** function determines if a task is executing while waiting for a previous blocking call to finish.

```
BOOL WINAPI WinCPICIsBlocking(void);
```

Return Values

The return value specifies the outcome of the function. If the value is not zero, there is an outstanding blocking call awaiting completion. A value of zero indicates the absence of an outstanding blocking call.

Remarks

This call does not infer any information about a particular conversation; it is only intended to provide help to an application written to use the CM_BLOCKING characteristic of Set_Processing_Mode. **WinCPICIsBlocking** serves the same purpose as **InSendMessage** in the Microsoft® Windows® API. Applications targeted at Windows version 3.*x* that support multiple conversations must specify CM_NONBLOCKING in **Set_Processing_Mode** so they can support multiple outstanding operations simultaneously. Applications are still limited to one outstanding operation per conversation in all environments.

Although a call issued on a blocking function appears to an application as though it blocks, the Windows CPI-C DLL has to relinquish the processor to allow other applications to run. This means that it is possible for the application that issued the blocking call to be re-entered, depending on the message(s) it receives. In this instance, **WinCPICIsBlocking** can be used to determine whether the application task currently has been re-entered while waiting for an outstanding blocking call to finish. Note that Windows CPI-C prohibits more than one outstanding blocking call per thread.

See Also

Specify_Windows_Handle, WinCPICSetBlockingHook, WinCPICUnhookBlockingHook

WinCPICSetBlockingHook

The **WinCPICSetBlockingHook** function allows a Windows CPI-C implementation to block CPI-C function calls by means of a new function. This call is used by Microsoft® Windows® version 3.*x* applications to make blocking calls without blocking the rest of the system. By default in the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows 98, and Microsoft® Windows 95 systems, blocking calls suspend the calling application's thread until the request is finished. Therefore, if a single-threaded application is targeted at both the Windows version 3.*x* system and the Windows 2000, Windows NT, Windows 98, and Windows 95 systems, and relies on this functionality, it should register a blocking hook even if the default hook will suffice.

```
FARPROC WINAPI WinCPICSetBlockingHook(
    FARPROC lpBlockFunc
   );
```

Parameters

lpBlockFunc
Specifies the procedure instance address of the blocking function to be installed.

Return Values

The return value points to the procedure instance of the previously installed blocking function. The application or library that calls **WinCPICSetBlockingHook** should save this return value so that it can be restored if needed. (If nesting is not important, the application can simply discard the value returned by **WinCPICSetBlockingHook** and eventually use WinCPICUnhookBlockingHook to restore the default mechanism.)

Remarks

A Windows CPI-C implementation has a default mechanism by which blocking CPI-C functions are implemented. This function gives the application the ability to execute its own function at blocking time in place of the default function.

The default blocking function is equivalent to:

```
BOOL DefaultBlockingHook (void)  {
    MSG msg;
    /* get the next message if any */
    if ( PeekMessage (&msg,0,0,PM_NOREMOVE)  )  {
        if ( msg.message = = WM_QUIT  )
            return FALSE;   // let app process WM_QUIT
        PeekMessage (&msg,0,0,PM_REMOVE) ;
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    /* TRUE if no WM_QUIT received */
    return TRUE;
}
```

The **WinCPICSetBlockingHook** function is provided to support applications that require more complex message processing—for example, those employing the multiple document interface (MDI) model or applications with Menu accelerators (TranslateAccelerator).

Blocking functions must return FALSE in response to a WM_QUIT message so Windows CPI-C can return control to the application to process the message and terminate gracefully. Otherwise, the function should return TRUE.

See Also

Set_Processing_Mode, Specify_Windows_Handle

WinCPICSetEvent

The **WinCPICSetEvent** function associates an event handle with a verb completion.

```
VOID WINAPI WinCPICSetEvent(
    unsigned char FAR * conversation_ID,
    HANDLE FAR * event_handle,
    CM_INT32 FAR *return_code
   );
```

Parameters

conversation_ID
Specifies the identifier for the conversation for which this event is used. This parameter is returned by the initial Accept_Conversation call.

event_handle
The handle of the event that is to be cleared when an asynchronous verb on the conversation completes. This parameter can replace an already-defined event or remove an already-defined event (by having NULL as the parameter).

return_code
The code returned from this call. The valid return codes are listed below.

Return Codes

CM_OK
The function executed successfully.

CM_PROGRAM_PARAMETER_CHECK
One or more of the parameters passed to this function are invalid.

CM_OPERATION_NOT_ACCEPTED
This value indicates that a previous operation on this conversation is incomplete and the **WinCPICSetEvent** call was not accepted.

Remarks

When a verb is issued on a nonblocking conversation, it returns CM_OPERATION_INCOMPLETE if it is going to complete asynchronously. If an event has been registered with the conversation, then the application can call **WaitForSingleObject** or **WaitForMultipleObjects** to be notified of the completion of the verb. When the verb has completed, the application must call Wait_for_Conversation to determine the return code for the asynchronous verb.

It is the responsibility of the application to reset the event, as it is with other APIs.

See also

Cancel_Conversation

WinCPICStartup

The **WinCPICStartup** function allows an application to specify the version of Windows CPI-C required and to retrieve details of the specific Windows CPI-C implementation. This function must be called by an application to register itself with a Windows CPI-C implementation before issuing any further Windows CPI-C calls.

```
INT WINAPI WinCPICStartup(
    WORD wVersionRequired,
    LPWCPICDATA lpwcpicdata
   );
```

Parameters

wVersionRequired
Specifies the version of Windows CPI-C support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

lpwcpicdata
A pointer to the CPI-C data structure. The **CPICDATA** structure is defined as follows:

```
typedef struct {
....WORD wVersion;
    char szDescription[WCPICDESCRIPTION_LEN+1];
}  CPICDATA, FAR * LPWCPICDATA;
```

where WCPIDESCRIPTION is defined to 127 and the structure members are as follows:

wVersion

The version of Windows CPI-C supported. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

szDescription

The description string describing the CPI-C version supported.

Return Values

The return value specifies whether the application was registered successfully and whether the Windows CPI-C implementation can support the specified version number. If the value is zero, it was registered successfully. Otherwise, the return value is one of the following:

WCPICSYSNOTRERADY
The underlying network subsystem is not ready for network communication.

WCPICVERNOTSUPPORTED
The version of Windows CPI-C support requested is not provided by this particular Windows CPI-C implementation.

WCPICINVALID
The Windows CPI-C version specified by the application is not supported by this DLL.

Remarks

To support future Windows CPI-C implementations and applications that may have functionality differences from Windows CPI-C version 1.0, a negotiation takes place in **WinCPICStartup**. An application passes to **WinCPICStartup** the Windows CPI-C version that it can use. If this version is lower than the lowest version supported by the Windows CPI-C DLL, the DLL cannot support the application and the **WinCPICStartup** call fails. If the version is not lower, however, the call succeeds and returns the highest version of Windows CPI-C supported by the DLL. If this version is lower than the lowest version supported by the application, the application either fails its initialization or attempts to find another Windows CPI-C DLL on the system.

This negotiation allows both a Windows CPI-C DLL and a Windows CPI-C application to support a range of Windows CPI-C versions. An application can successfully use a DLL if there is any overlap in the versions. The following table illustrates how **WinCPICStartup** works in conjunction with different application and DLL versions.

Application versions	DLL versions	To WinCPICStartup	From WinCPICStartup	Result
1.0	1.0	1.0	1.0	Use 1.0
1.0, 2.0	1.0	2.0	1.0	Use 1.0
1.0	1.0, 2.0	1.0	2.0	Use 1.0

1.0	2.0, 3.0	1.0	WCPICINVALID	Fail
2.0, 3.0	1.0	3.0	1.0	App Fails
1.0, 2.0, 3.0	1.0, 2.0, 3.0	3.0	3.0	Use 3.0

Details of the actual Windows CPI-C implementation are described in the **WHLLDATA** structure defined as follows:

```
typedef struct tagWCPICDATA { WORD wVersion;
            char szDescription[WHLLDESCRIPTION_LEN+1];
            } WCPICDATA, FAR *LPWCPICDATA;
```

Having made its last Windows CPI-C call, an application should call the WinCPICCleanup routine.

Each Windows CPI-C implementation must make a **WinCPICStartup** call before issuing any other Windows CPI-C calls.

WinCPICUnhookBlockingHook

The **WinCPICUnhookBlockingHook** function removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

```
BOOL WINAPI WinCPICUnhookBlockingHook(void);
```

Return Values

The return value specifies the outcome of the function. It is not zero if the default mechanism is successfully reinstalled. The value is zero if the mechanism did not reinstall.

See also

WinCPICSetBlockingHook

Common Return Codes

This section describes the return codes for CPI-C calls. The return codes are listed in integer order.

Call-specific return codes are described for the individual calls in CPI-C Calls.

0

CM_OK
The call executed successfully.

1

CM_ALLOCATION_FAILURE_NO_RETRY
The conversation cannot be allocated because of a permanent condition, such as a configuration error or session protocol error. To determine the error, the system administrator should examine the error log file. Do not retry the allocation until the error has been corrected.

2

CM_ALLOCATION_FAILURE_RETRY
The conversation could not be allocated because of a temporary condition, such as a link failure. The reason for the failure is logged in the system error log. Retry the allocation.

3

CM_CONVERSATION_TYPE_MISMATCH
The partner LU or program does not support the conversation type (basic or mapped) specified in the allocation request.

5

CM_PIP_NOT_SPECIFIED_CORRECTLY
The allocation request was rejected by a non-CPI-C LU 6.2 TP. The partner program requires one or more PIP data variables, which are not supported by CPI-C.

6

CM_SECURITY_NOT_VALID
The user identifier or password specified in the allocation request was not accepted by the partner LU.

8

CM_SYNC_LVL_NOT_SUPPORTED_PGM
The partner program does not support the synchronization level specified in the allocation request.

9

CM_TPN_NOT_RECOGNIZED
The partner LU does not recognize the program name specified in the allocation request.

10

CM_TP_NOT_AVAILABLE_NO_RETRY
The partner LU cannot start the program specified in the allocation request because of a permanent condition. The reason for the error may be logged on the remote node. Do not retry the allocation until the error has been corrected.

11

CM_TP_NOT_AVAILABLE_RETRY
The partner LU cannot start the program specified in the allocation request because of a temporary condition. The reason for the error may be logged on the remote node. Retry the allocation.

17

CM_DEALLOCATED_ABEND
The conversation has been deallocated for one of the following reasons:

- The remote program issued Deallocate with the type parameter set to CM_DEALLOCATE_ABEND. If the conversation for the remote program was in RECEIVE state when the call was issued, information sent by the local program and not yet received by the remote program is purged.

- The partner program terminated normally but did not deallocate the conversation before terminating.

18

CM_DEALLOCATED_NORMAL
This return code does not indicate an error.

The partner program issued the Deallocate call with *deallocate_type* set to one of the following:

- CM_DEALLOCATE_FLUSH.
- CM_DEALLOCATE_SYNC_LEVEL with the synchronization level of the conversation specified as CM_NONE.

19

CM_PARAMETER_ERROR
The local program specified an invalid argument in one of its parameters.

20

CM_PRODUCT_SPECIFIC_ERROR
A product-specific error occurred and has been logged in the product's error log.

21

CM_PROGRAM_ERROR_NO_TRUNC
While in SEND state or in SEND-PENDING state with the error direction set to CM_SEND_ERROR, the partner program issued Send_Error. Data was not truncated.

22

CM_PROGRAM_ERROR_PURGING
One of the following occurred:

- •While in RECEIVE or CONFIRM state, the partner program issued Send_Error. Data sent but not yet received is purged.
- •While in SEND-PENDING state with the error direction set to CM_RECEIVE_ERROR, the partner program issued **Send_Error**. Data was not purged.

23

CM_PROGRAM_ERROR_TRUNC (for a basic conversation)
In SEND state, before finishing sending a complete logical record, the partner program issued Send_Error. The local program may have received the first part of the logical record through a Receive call.

24

CM_PROGRAM_PARAMETER_CHECK
A parameter or the address of a variable is invalid. For details, see individual calls in CPI-C Calls.

25

CM_PROGRAM_STATE_CHECK
The call was not issued in an allowed conversation state. For details, see individual calls in CPI-C Calls.

26

CM_RESOURCE_FAILURE_NO_RETRY
One of the following occurred:

- The conversation was terminated prematurely because of a permanent condition. Do not retry until the error has been corrected.
- The partner program did not deallocate the conversation before terminating normally.

27

CM_RESOURCE_FAILURE_RETRY
The conversation was terminated prematurely because of a temporary condition, such as modem failure. Retry the conversation.

28

CM_UNSUCCESSFUL

The verb issued by the local program was not executed successfully.

30

CM_DEALLOCATED_ABEND_SVC
The conversation has been deallocated for one of the following reasons:

- The partner program issued Deallocate with the type parameter set to ABEND_SVC.
- The partner program did not deallocate the conversation before terminating.

If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

31

CM_DEALLOCATED_ABEND_TIMER
The conversation has been deallocated because the partner program issued Deallocate with the type parameter set to ABEND_TIMER. If the conversation is in RECEIVE state for the partner program when this call is issued by the local program, data sent by the local program and not yet received by the partner program is purged.

32

CM_SVC_ERROR_NO_TRUNC (for a basic conversation)
While in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data was not truncated.

33

CM_SVC_ERROR_PURGING
While in SEND state, the partner program or partner LU issued Send_Error with the type parameter set to SVC. Data sent to the partner program may have been purged.

34

CM_SVC_ERROR_TRUNC (for a basic conversation)
While in RECEIVE or CONFIRM state, the partner program or partner LU issued Send_Error with the type parameter set to SVC before it finished sending a complete logical record. The local program may have received the first part of the logical record.

35

CM_OPERATION_INCOMPLETE
The operation has not completed and is still in progress. The program can issue Wait_For_Conversation to await the completion of the operation, or Cancel_Conversation to cancel the operation and conversation. If Specify_Windows_Handle has been called, the application should wait for notification by a windows message and not call **Wait_For_Conversation**.

36

CM_SYSTEM_EVENT
This error code is not used by Microsoft® Host Integration Server or Microsoft® SNA Server.

37

CM_OPERATION_NOT_ACCEPTED
A previous operation on this conversation is incomplete.

CPI-C Sample Applications

This section of the Microsoft® Host Integration Server 2000 Developer's Guide describes the sample programs that implement the CPI-C.

This section contains:

- Sample CPI-C TPs in the SDK

Sample CPI-C TPs in the SDK

The source code for several sample programs that illustrate using CPI-C for transaction programs (TPs) are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\SNA subdirectory on the Host Integration Server 2000 CD-ROM (these samples are located under the \SDK\SAMPLES folder on earlier versions of SNA Server). These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\SNA subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\Sna subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files:

Sample TP program	Description
APING and APINGD	Sample programs that provide a simple test for end-to-end connectivity. These samples are located i n the \SDK\Samples\SNA\aping folder on the CD-ROM.
Multithreaded APINGD	A multithreaded connectivity test that illustrates nonqueued behavior in Microsoft® Window 2000, Microsoft® Window NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95. This sample i s located in the \SDK\Samples\SNA\mping folder on the CD-ROM.
CPI-C Send and Receive TPs	A pair of simple CPI-C TPs that illustrate the use of asynchronous CPI-C calls. These samples are locat ed in the \SDK\Samples\SNA\cpic folder on the CD-ROM.
AREXEC and AREXECD	A pair of TPs that execute commands on a remote computer and send the output back across the con nection. These samples are located in the \SDK\Samples\SNA\arexec folder on the CD-ROM.
AREMOTE	A sample client and server program using APPC that enables you to invoke and control a text-mode program from another computer. This sample using APPC was based a Win32® sample program th at originally used named pipes. These samples are located in the \SDK\Samples\SNA\aremote folder on the CD-ROM.

In addition to these TPs, the following supplemental programs are included on the Host Integration Server 2000 CD-ROM.

Supplemental program	Description
TPSETUP	A sample installation program, demonstrating an interface that assists in configuring autostarted invokable TPs. This sa mple is located in the \SDK\Samples\SNA\tpsetup folder on the CD-ROM.
TPSTART	A program required for the automatic startup of invokable TPs that run as applications under Microsoft® Window 2000 or Window NT. TPSTART is not required if the TP has been written as a Window 2000 or Window NT service. TPSTART i s also unnecessary under Window 98 and Window 95. TPSTART is installed by Host Integration Server 2000 Setup in th e System folder of the Host Integration Server 2000 root directory. This sample is located in the \SDK\Samples\SNA\tps tart folder on the CD-ROM.

Building the TPs

The CPI-C samples are designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio .NET interactive development environment (IDE).

To build the CPI-C samples installed from the Host Integration Server CD-ROM, set the following environment variables:

Variable	Description
ISVLIBS	The directory containing the Microsoft® Host Integration Server 2000 LIB files for Microsoft® Windows 2000, Microsoft® Window NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95.
ISVINCS	The directory containing the Host Integration Server 2000 header files.
SAMPLEROOT	The root directory where the sample code provided as part of the SDK has been installed on a local hard disk.

For example, if you installed the Host Integration Server SDK directory to the default location (C:\Program Files\Host Integration Server SDK), use the following lines to set the variables (assumes Intel binaries are being produced for Windows 2000, Windows NT on I386, Windows 98, or Windows 95) :

```
ISVLIBS=C:\Program Files\Host Integration Server SDK\LIB
ISVINCS=C:\Program Files\Host Integration Server SDK\Include
SAMPLEROOT=C:\Program Files\Host Integration Server SDK\Samples\SNA
```

Change to each subdirectory and run NMAKE on the .MAK file in each directory. For example, for the APING and APINGD sample, change to the aping subdirectory and type the following:

nmake -f makeping.mak

Note that Windows NT on DEC Alpha is not supported by the Host Integration Server SDK. If you wish to build these samples on Windows NT 4.0 for DEC Alpha, the earlier SNA Server 4.0 SDK will be required for accessing the Windows NT import libraries for DEC Alpha under the \SDK\LIB\WINNT\ALPHA folder.

To build the CPI-C samples installed as part of the MSDN Platform SDK using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory. The default location of this file is C:\Program Files\Microsoft Visual Studio\VC98\Bin (for VS6) or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools (for VS.NET)

To build all the SNA samples, open an MS-DOS Command Prompt window, navigate to the SNA subdirectory, and invoke NMAKE. This will recursively invoke NMAKE and build all of the SNA samples including the APPC samples.

To build a specific sample (APING or APINGD, for example) using the command-line compiler, open an MS-DOS Command Prompt window, navigate to the appropriate subdirectory (SNA\aping, for example), and invoke NMAKE.

To build a specific sample (APING, for example) using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (SNA\aping\aping.vcproj, for example) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

TPSETUP

The TPSETUP sample program simplifies the setting of registry or environment variables needed by autostarted invokable TPs. Without an interface provided by TPSETUP, configuring such variables can be complicated and prone to errors. Therefore, it is recommended that you use code like TPSETUP in installation programs for autostarted invokable TPs.

Operation

The source code for TPSETUP, contained in INSTALL.C, can be compiled to work in the Window 2000, Window NT, Window 98, and Window 95 environment or in the Windows version 3.*x* environment. TPSETUP has been constructed so that the program responds correctly in each environment.

For clients running Window 2000 and Window NT, it is recommended that autostarted invokable TPs be written as Window 2000 or Window NT services. For the installation program for such TPs, study the code in INSTALL.C. For example, use the **CreateService** function or similar code when installing a TP that will run as a service under Window 2000 or Window NT. For important information about how services work under Window 2000 and Window NT, see the documentation included with the Platform SDK for Window 2000, Window NT, and Win32®.

TPSTART

An autostarted TP that runs as an application under Window 2000 or Window NT requires the support of the TPSTART program, which is installed with the Host Integration Server 2000 software in the System subdirectory of the Host Integration Server 2000 root directory. Therefore, the TPSTART program must be started on a Window 2000-based or Window NT-based client before an autostarted TP can start as an application. Starting TPSTART can be accomplished by using standard Window 2000 or Window NT methods, such as including TPSTART in the **Startup** group on the client.

APING and APINGD

The sample code for APING and APINGD is ported from code on the IBM APPC/CPI-C disk. These samples are used to test end-to-end connectivity; and simply show that the configuration is correct by exchanging bytes of data across the link. APING is the client or invoking (local) half and attempts to contact APINGD (the APPC/CPI-C ping daemon or server), which is written here as a Window 2000 or Window NT service so it can be installed as an invokable TP on the remote computer.

Setup

To set up APING and APINGD

1. Create an appropriate APPC LU-LU-mode triplet (for example, LUPING-LUPINGD-#INTER).
2. Set up a CPI-C symbolic destination name that contains the configured remote LU and mode. (The TP name for APINGD is APINGD.)
3. Assign the local APPC LU to the APING TP, either by using a registry entry of APING:REG_SZ:*LocalLUAlias* in the **SnaBase\Parameters\Clients** key, or by assigning the local LU as the default local APPC LU for the user who will run APING.

Input and Output

APING is a console application. The syntax of its command line is

aping [**-s***size*] [**-i***iterations*] [**-c***packets*] [**-m***mode*] [**-t***tpname*] *PartnerLUName*

aping [**-s***size*] [**-i***iterations*] [**-c***packets*] *SymbolicDestinationName*

where

-s*size*
Specifies the size, in bytes, of the packet transmitted. The default is 100 bytes.
-i*iterations*
Specifies the number of iterations to carry out. The default is 2.
-c*packets*
Specifies the number of consecutive packets sent by each side. The default is 1.
-m*mode*
Specifies the mode name. The default is #INTER.
-t*tpname*
Specifies the TP name of the TP to start on the remote server. The default is APINGD.
PartnerLUName
Specifies the partner LU name of the destination.
SymbolicDestinationName
Specifies the symbolic destination name of the destination.

Output goes to **stdout** and **stderr**, and details the data rates and timings for each iteration.

Operation

Note that with APINGD, Specify_Local_TP_Name is used to set the local TP name, so Wait_For_Conversation must be used to wait for the Accept_Conversation call to complete, because it will return asynchronously.

The code at the end of APINGD.C is a stub for making any TP into a Window 2000 or Window NT service. There are three routines that are needed: **main**, **ServiceMain**, and **ControlHandler**. See the comments in the file for details of how these work. The **TPStart** routine is the entry point of the TP proper.

In particular, note that in response to the SERVICE_CONTROL_STOP or SERVICE_CONTROL_SHUTDOWN messages in the **ControlHandler** routine, action should normally be taken to stop the service, but because each run does not last long with these samples, no code is included to take such an action.

Multithreaded APINGD

The version of MPINGD provided in the sample code illustrates how to achieve nonqueued behavior from an invokable TP in Window 2000, Window NT, Window 98, or Window 95. This means that multiple copies of APING can talk to the same copy of MPINGD at the same time. However, you cannot run multiple copies of a Window 2000 or Window NT service. The features are achieved by always having a thread with an Accept_Conversation outstanding, so that any incoming attach for MPINGD will always be satisfied immediately.

Setup

Setup requirements for MPINGD are the same as for the single-threaded version, APINGD. The remote LU and mode that you use should support parallel sessions so that more than one conversation at a time is possible.

Input and Output

The input and output for MPINGD are the same as for the single-threaded version, APINGD.

Operation

The operation of MPINGD is similar to that of the single-threaded version, APINGD. The same three routines are used (**main**, **ServiceMain** and **ControlHandler**). **ServiceMain** calls the **TPStart** routine. This routine must not return until the service is ready to terminate.

The **TPStart** routine does some initialization, creates the first conversation thread, and then waits on an event created by the **ServiceMain** routine. This event is set when the service control manager issues an order to STOP or SHUTDOWN. When the event is set, it calls WinCPICCleanup, which will cancel any active conversations and return outstanding Accept_Conversation calls, thus making all conversation threads exit. It then marks the service as STOPPED.

The **ThreadStart** routine is the entry point for each of the conversation threads. It issues **Accept_Conversation** and Wait_For_Conversation, and when this completes, it creates another thread to wait for the next attach while the existing thread services the first attach.

CPI-C Send and Receive TPs

These TPs are CPI-C versions of the APPC send and receive TPs. The sample code illustrates the use of asynchronous CPI-C calls.

Setup

To set up the send and receive TPs

1. Create an appropriate APPC LU-LU-mode triplet.
2. Set up a CPI-C symbolic destination name that contains the configured remote LU and mode. (The default symbolic destination name is CPICRECV.)
3. Assign the local APPC LU to the CPICSEND TP, either by using a registry entry of CPICSEND:REG_SZ:*LocalLUAlias* in the **SnaBase\Parameters\Clients** key, or by assigning the local LU as the default local APPC LU for the user who will run CPICSEND.

For example, use SENDLU-RECVLU-#INTER as your LU-LU-mode triplet. Then create a CPI-C symbolic destination name CPICRECV containing the application TP name CPICRECV, the partner LU alias RECVLU, and the mode name #INTER. Finally, add the intended user to the users list, and assign SENDLU as the user's default local APPC LU.

Input and Output

CPICSEND and CPICRECV use the files Cpicsend.cfg and Cpicrecv.cfg for input. These files should be placed in the folder that contains the TP executable file. These files are similar to the input files for the APPC send and receive TPs.

The following entries are for CPICSEND only:

Line	Default Value	Description
ResultFile =	C:\Cpicsend.out	The filename where the timings results will be stored.
NumSends =	2	The number of Send_Data calls per conversation.
SendSize =	1024	The size of data sent each time in bytes .
ConfirmEvery =	1	The number of **Send_Data** calls between Confirm calls. If ConfirmEvery=0, then CPICSEND will not issue CONFIRM verbs.
SymDestName =	CPICRECV	The symbolic destination name.
NumConversations =	1	The number of conversations. This setting must be the same for CPICSEND and CPICRECV (they do not negotiate the number). If this value is zero, then the TPs will do an infinite number of conversations.
WaitMode=	No	Yes, No, or Block. If WaitMode=No, the verbs are completed through posted windows messages. The TPs issue Specify_Windows_Handle with a window handle so that Windows CPI-C will post completion messages to this window handle. If WaitMode=Yes, verbs are non-blocking and completed using asynchronous call completion. In this case, the TPs issue **Specify_Windows_Handle** with NULL so that the TPs must then issue a Wait_For_Conversation call to wait for the asynchronous call to complete. If WaitMode=Block, all verbs are blocking.

The following entries are for CPICRECV only:

Line	Default Value	Description

ResultFile =	C:\Cpicrecv.out	The filename where the timings results will be stored.
LocalTPName =	CPICRECV	The local TP name to use on the Specify_Local_TP_Name call.
NumConversations =	1	The number of conversations. This setting must be the same for CPICSEND and CPICRECV (they do not negotiate the number). If this value is zero, then the TPs will do an infinite number of conversations.
WaitMode=	No	Yes, No, or Block. If WaitMode=No, the verbs are completed through posted windows messages. The TPs issue Specify_Windows_Handle with a window handle so that Windows CPI-C will post completion messages to this window handle. If WaitMode=Yes, verbs are non-blocking and completed using asynchronous call completion. In this case, the TPs issue **Specify_Windows_Handle** with NULL so that the TPs must then issue a Wait_For_Conversation call to wait for the asynchronous call to complete. If WaitMode=Block, all verbs are blocking.

As with CPICSEND, CPICRECV produces C:\Cpicrecv.out (by default) with timings of the conversations in it.

Operation

CPICRECV should be started first; it issues Specify_Local_TP_Name to set its local TP name, and then Accept_Conversation to accept a conversation (note that because **Specify_Local_TP_Name** is issued, the **Accept_Conversation** will complete asynchronously).

Both TPs issue Specify_Windows_Handle during initialization to set either the window handle or NULL CPICSEND calls Set_Processing_Mode after completion of Initialize_Conversation to set the processing mode to nonblocking for this conversation.

After each call is issued, the return code is checked; if it is not CM_OPERATION_INCOMPLETE, the call has already completed, so an ASYNC_COMPLETE message is posted to trigger the next call. If *WaitMode* is set to YES and the issued call did not complete immediately, then a Wait_For_Conversation call is issued to wait for call completion, at which point an ASYNC_COMPLETE message is posted. If *WaitMode* is set to NO and the issued call did not complete immediately, then Windows CPI-C detects call completion and posts an ASYNC_COMPLETE message. The receipt of the ASYNC_COMPLETE message triggers the next call to be issued.

For CPICSEND, each conversation consists of an Allocate call, followed by a given number of Send_Data calls of given size and interspersed with Confirm calls at a given interval, followed by a Deallocate.

CPICRECV issues Receive on completion of the Accept_Conversation, and then issues either **Receive** or Confirmed according to the return from the previous **Receive**.

At any stage, if the TPs encounter an error, they terminate. Use CPI-C API tracing to diagnose problems with the configuration.

Both TPs are built from a single source-code file, CPICSR.C. CPICSEND is compiled only if CPICSEND macro is #defined. This macro is normally defined using the -DCPICSEND option on the command line to the C compiler.

The TPs run as Window 2000, Window NT, Windows 98, or Window 95 applications with a minimized window, the title of which displays the status. When the WndProc of this window, TPWndProc, receives the WM_CREATE message for the window, it triggers the issuing of the first call. When TPWndProc receives an ASYNC_COMPLETE message from Windows CPI-C, it triggers the issuing of the next call, dependent on what the previous call was. When the window is closed, WinCPICCleanup is issued to terminate any active conversations.

AREXEC and AREXECD

The sample code for these two TPs provides the ability to execute commands on a remote computer and to send the output back across the connection to the invoking TP.

Setup

To set up AREXEC and AREXECD

1. Create an appropriate APPC LU-LU-mode triplet.
2. Set up a CPI-C symbolic destination name that contains the configured remote LU and mode. (The TP name for AREXECD is AREXECD.)
3. Assign the local APPC LU to the AREXEC TP, either by using a registry entry of AREXEC:REG_SZ:*LocalLUAlias* in the **SnaBase\Parameters\Clients** key, or by assigning the local LU as the default local APPC LU for the user who will run AREXEC.

Input and Output

AREXEC is a console application. The syntax of its command line is

arexec [**-m***mode*] [**-t***tpname*] *destination command*

where

-m*mode*
Specifies the mode name. The default is #INTER.
-t*tpname*
Specifies the TP name.
destination
Specifies the destination. Can be either a symbolic destination name or a partner LU name.
command
Specifies the command string to execute on the remote computer.

The **stdout** and **stderr** from the command executed at the remote end is sent across the link and printed to **stdout** on the invoking end.

Operation

The AREXECD program is a Window 2000 or Window NT service, using the same routines in APING, APINGD and multithreaded APINGD. The execution of the command and sending back of data are done in the routine **execute_and_send_output** in CPICPORT.C. This sample creates a named pipe and connects to the read end of the pipe. It then creates a process to run the command and gives that process a handle to the write end of the pipe as its **stdout** and **stderr**. Then the data is read from the pipe, and Send_Data is used to send it across the link.

AREMOTE

The sample code for this TP provides the ability to control a text-mode program on a remote computer. AREMOTE is a Win32 console application that implements a client and a server. The AREMOTE server is invoked with the name of the text-mode program that the client wishes to control remotely. The AREMOTE client redirects **stdin** (keyboard input) from the client to the AREMOTE server. In turn, the AREMOTE server redirects **stdin** and **stderr** from the program being controlled back to the AREMOTE client.

Setup

To set up AREMOTE

1. Create an appropriate APPC LU-LU-mode triplet.
2. Set up a CPI-C symbolic destination name that contains the configured remote LU and mode. (The TP name for AREMOTE is AREMOTE.)
3. Assign the local APPC LU to the AREMOTE TP, either by using a registry entry of AREMOTE:REG_SZ:*LocalLUAlias* in the **SnaBase\Parameters\Clients** key, or by assigning the local LU as the default local APPC LU for the user who will run AREMOTE.

Input and Output

The syntax of the command line to start the client end of AREMOTE is as follows:

aremote /C *ServerLU* [**/T** *TPName*] [**/P** *TPName*] [**/L** *LocalLU*]
[**/M** *Modename*] [**/N** *Lines*] [**/F** *Color*] [**/B** *Color*]

where

/C
Specifies the client mode.
ServerLU
Specifies the SNA LU for connecting to the server.
/T *TPName*
Specifies the TP name that the server is using. The default is AREMOTE.
/P *TPName*
Specifies the TP name that the client is using. The default is AREMOTC.
/L *LocalLU*
Specifies the LU name for the local TP to use. The default is AREMOTE.
/M *Modename*
Specifies the mode name. The default is #INTER.
/N *Lines*
Specifies the number of lines to get.
/F *Color*
Specifies the foreground color. Color options are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lcyan, lred, lpurple, lyellow, and lwhite.
/B *Color*
Specifies the background color. Color options are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lcyan, lred, lpurple, lyellow, and lwhite.

The syntax of the command line to start the server end of AREMOTE is as follows:

aremote /S *Cmd* [**/T** *TPName*] [**/M** *Modename*] [**/F** *Color*] [**/B** *Color*]

where

/S
Specifies the server mode.
Cmd
Specifies a text-mode program that you want to control from another computer.
/T *TPName*
Specifies the TP name that the server is using. The default is AREMOTE.
/M *Modename*
Specifies the mode name. The default is #INTER.
/F *Color*
Specifies the foreground color. Color options are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lcyan,

lred, lpurple, lyellow, and lwhite.

/B *Color*

Specifies the background color. Color options are black, blue, green, cyan, red, purple, yellow, white, lblack, lblue, lgreen, lcyan, lred, lpurple, lyellow, and lwhite.

The **stdout** and **stderr** from the command run at the remote end is sent across the link and printed to **stdout** on the client. The **stdin** from the client is sent across the link and becomes the **stdin** for the command run at the remote end.

The APPC remote installer (ARSETUP) included with this sample brings up a dialog box that prompts for TP configuration information. The information is then placed in the registry under Window 2000, Window NT, Window 98, and Window 95 or in the Win.ini file under Windows 3.x. The WIN32 compiler flag specifies that the Win32 version of ARSETUP should be built for use on Window 2000, Window NT, Window 98, and Window 95. The WINDOWS flag specifies that the Windows 3.x version of ARSETUP should be built.

Operation

The AREMOTE server can also be configured to run as a Window 2000 or Window NT service using the ARSETUP sample utility included in the same folder on the CD-ROM.

The AREMOTE client can exit by inputting the following character sequences:

%cQ : Quit but leave the AREMOTE server running.

%cK : Exit and stop the AREMOTE server.

Other special client commands include the following:

%cM : Send a message to the AREMOTE server.

%cP : Show a popup on the AREMOTE server.

%cS : Report the status of the AREMOTE server.

%cH : Provide help describing these special client commands.

LUA Applications

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop C-language applications that use the conventional Logical Unit Application programming interface (LUA) to exchange data in a Systems Network Architecture (SNA) environment.

This section contains:

About the LUA Guide

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop C-language applications that use the conventional Logical Unit Application programming interface (LUA) to exchange data in a Systems Network Architecture (SNA) environment.

This implementation of LUA is compatible with the Request Unit Interface (RUI) and the Session Level Interface (SLI) of LUA for the Microsoft® Windows NT® and Microsoft Windows 95 operating systems, the Windows graphical environment, and the IBM Extended Services for OS/2 version 1.0.

This guide provides conceptual information and detailed reference information.

To use this guide effectively, you should be familiar with:

- Microsoft Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows® 2000
    - Microsoft Windows NT®
    - Microsoft Windows 98
    - Microsoft Windows 95
- SNA concepts

This guide provides conceptual information and detailed reference information. Before reading it, you should be familiar with:

- General concepts for the communications software you have installed. (Refer to your product documentation for information.)
- SNA concepts.
- Microsoft C version 5.1 or later.
- One of the following operating environments:

  Microsoft Windows NT

  Microsoft Windows 95 or Windows 98

  Microsoft Windows version 3.0 or later

This section contains

Operating Systems Support for LUA Development

This section of the guide contains information relating to following operating systems:

Microsoft® Windows® 2000

Microsoft Windows NT®

Microsoft Windows® 98

Microsoft Windows® 95

Microsoft Windows version 3.*x*

Microsoft MS-DOS®

OS/2 (with IBM Extended Services for OS/2 version 1.0)

Microsoft Host Integration Server 2000 supports the development of LUA applications for Windows 2000, Windows NT, Windows 98, and Windows 95. Under these operating systems, support for LUA applications is provided only for the Win32® subsystem.

The previous Microsoft SNA Server product also supported the development of LUA applications for Windows 3.*x* and OS/2. Most LUA applications developed for Windows 3.x and OS/2 with SNA Server can be used with Host Integration Server 2000. The Windows 3.x, MS-DOS, and OS/2 interface is described here for completeness, but Windows 3.x, MS-DOS, or OS/2 LUA application development is not supported using Host Integration Server.

Finding Further Information

This guide does not provide a detailed explanation of products, architectures, or standards other than those directly pertaining to Windows LUA. For information on specific operating environments, refer to your system documentation. For information about SNA, refer to your system network documentation.

For information about SNA architecture, refer to your system network documentation.

The following documents provide additional information about Host Integration Server application programming interfaces (APIs) based on SNA architecture:

- APPC Applications section of the *Microsoft Host Integration Server Developer's Guide*
- CPI-C Applications section of the *Microsoft Host Integration Server Developer's Guide*

For more information about SNA and about 3270 information display systems, see the following manuals:

IBM 3270 Information Display System: 3274 Control Unit Description and Programmer's Guide

IBM 3270 Information Display System: Color and Programmed Symbols

IBM 3270 Information Display System: 3274 Control Unit Display Station: Operator's Guide

IBM Systems Network Architecture: Technical Overview

IBM Systems Network Architecture: Concepts and Products

IBM Advanced Communications Function Products Installation Guide

IBM Installation and Resource Definition

IBM 9370 LAN Token Ring Support

IBM SNA Format and Protocol Reference Manual: Architectural Logic

For background information about logical unit (LU) 6.2, Advanced Program-to-Program Communications (APPC), and/or the Common Programming Interface for Communications (CPI-C), see the following manuals:

IBM Systems Network Architecture: Introduction to APPC

IBM Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2

IBM SNA: Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2

IBM SNA: Formats

IBM SNA: Technical Overview

IBM SNA: ACF/VTAM Programming for LU Type 6.2

LUA Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides the programmatic techniques and procedures for creating applications with the Logical Unit Application programming interface (LUA).

This section contains:

- LUA Concepts
- Writing LUA Applications
- Support for LUA Single Sign-On

LUA Concepts

The conventional Logical Unit Application (LUA) programming interface is an application programming interface (API) that allows you to write LUA applications to communicate with host applications.

The interface is provided at the request/response unit and session levels, allowing programmable control over the Systems Network Architecture (SNA) messages sent between your communications software and the host. It can be used to communicate with any of the logical unit types 0, 1, 2, or 3 at the host; the application must send the appropriate SNA messages as required by the host.

For example, you can use LUA to write a 3270 emulation program that communicates with a host 3270 application.

This section contains:

Windows LUA Overview

To provide one common API to port applications from various operating environments to Microsoft® Windows® 2000, Windows NT®, Windows 98, Windows 95, and Windows version 3.*x*, a Windows SNA standard was created. As a direct result of this work, Windows LUA was developed. The LUA verbs, routines, and information presented in this guide represent an evolving Windows LUA that is based on IBM Extended Services for OS/2 version 1.0 and includes a set of Windows extensions.

The use of the Windows LUA interface on Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2 will cause additional threads to be created within the calling process. These other threads perform interprocess communication with the SNA Server service over the LAN interface that the client is configured to use (TCP/IP, IPX/SPX, or named pipes, for example).

If an application using Windows LUA is running on Windows 2000 or Windows NT, stopping the SNABASE service will cause the application to be unloaded from memory.

This section contains:

Windows LUA Asynchronous Support

Asynchronous verb completion returns immediately from issuing an initial verb (before results have been received) so the application can continue with other processes. A program that issues a verb and does not regain control until the operation completes cannot perform any other operations. This synchronous type of operation, called blocking, is not suited to a server application designed to handle multiple requests from many clients.

By design, LUA is asynchronous and uses semaphores for notification messages. Semaphores work well for the Windows 2000, Windows NT, Windows 98, Windows 95, and OS/2 systems, and the Windows extensions built into Windows LUA also allow asynchronous support for Windows version 3.*x* applications in the LUA interface. Windows LUA provides the following functions for issuing the Request Unit Interface (RUI) and Session Level Interface (SLI) verbs:

RUI

SLI

WinRUI

WinSLI

WinRUI and **WinSLI** provide asynchronous message notification for all Windows-based RUI and SLI verbs, while **RUI** and **SLI** provide support for event notification. Windows version 3.*x* applications use **WinRUI** and **WinSLI** for asynchronous message notification.

Asynchronous support allows you to be notified of verb completion based on a window handle. You can register a window handle using the RegisterWindowsMessage function with "WinRUI" or "WinSLI" as the string. You then issue a verb using the WinRUI or WINSLI function and passing a window handle. When the LUA verb conversation completes, a message is posted to the window handle that you passed, notifying you that the verb is complete.

The only other Windows extension functions required for Windows LUA are for initialization (WinRUIStartup or WinSLIStartup) and termination (WinRUICleanup or WinSLICleanup) purposes. Depending on your application, other Windows extensions may be useful, but they are not required. A complete description of all Windows LUA verbs, routines, and extensions is provided in LUA RUI Verbs, LUA SLI Verbs, and LUA Extensions for the Windows Environment.

Before Using Windows LUA

The following Windows extensions are of particular importance and should be reviewed before using the LUA API and this version of Host Integration Server or the earlier SNA Server product:

- RUI

  Provides event notification for all RUI verbs. The application must provide a handle to an event in the **lua_post_handle** member of the verb control block (VCB). The event must be in the not-signaled state. When the asynchronous operation is complete, the application is notified through the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions.

- SLI

  Provides event notification for all SLI verbs. The application must provide a handle to an event in the **lua_post_handle** member of the VCB. The event must be in the not-signaled state. When the asynchronous operation is complete, the application is notified through the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions.

- WinRUI

  Provides asynchronous notification for all Windows-based RUI verbs. When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinRUI" as the input string. The *lParam* argument of the message contains the address of the VCB being posted as complete. The *wParam* argument of the message is undefined.

  An application must call WinRUIStartup for initialization before calling WinRUI.

- WinRUICleanup

  An application must call this function when finished using RUI verbs to deregister itself from the Windows LUA implementation. This function terminates and deregisters an application from a Windows LUA implementation.

- WinRUIStartup

  An application must call this function to register itself with a Windows LUA implementation before issuing any further Windows LUA calls using RUI verbs. This function allows an application to specify the version of Windows LUA required and to retrieve details of the specific LUA implementation.

- WinSLI

  Provides asynchronous notification for all Windows-based SLI verbs. When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinSLI" as the input string. The *lParam* argument of the message contains the address of the VCB being posted as complete. The *wParam* argument of the message is undefined.

  An application must call WinSLIStartup for initialization before calling WinSLI.

- WinSLICleanup

  An application must call this function when finished using SLI verbs to deregister itself from the Windows LUA implementation. This function terminates and deregisters an application from a Windows LUA implementation.

- WinSLIStartup

  An application must call this function to register itself with a Windows LUA implementation before issuing any further Windows LUA calls using SLI verbs. This function allows an application to specify the version of Windows LUA required and to retrieve details of the specific LUA implementation.

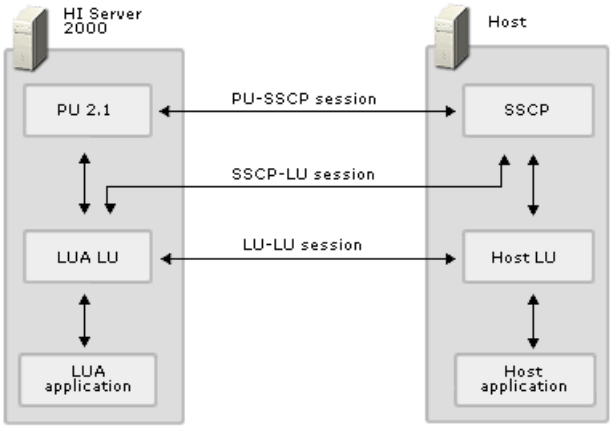
Using LUA and Asynchronous Verb Completion

Host Integration Server and SNA Server permit one outstanding Windows SNA asynchronous call per connection and one blocking verb per thread. When the asynchronous verb completes, LUA does the following depending on your environment:

- For a Windows 2000, Windows NT, Windows 98, or Windows 95 system, two types of notification are possible. The preferred type is the event method, in which the LUA application issues **WaitForSingleObject/WaitForMultipleObject**. The application can also post the "WinRUI"/"WinSLI" notification message to the window handle of the **WinRUI/WinSLI** message.
- For a Windows version 3.*x* system, LUA notifies the completion of an asynchronous request by posting the "WinRUI"/"WinSLI" notification message to the window handle of the WinRUI/WinSLI message. A window handle has been added as the first parameter passed to the WinRUI and WinSLI functions.
- For OS/2, LUA notifies the completion of an asynchronous request by posting a message to the application's Window procedure or by clearing a semaphore/event. Additionally, the application can use system queues for asynchronous verbs.

LUs and Sessions

The following figure shows the SNA components required for LUA communications.



An LUA application uses a local LU, which uses Host Integration Server or SNA Server to communicate with the host system. There are three progressive sessions when Host Integration Server or SNA Server connects to the host node:

- The PU-SSCP session, between the Host Integration Server or SNA Server physical unit (PU) and the host's system services control point (SSCP); this is used mainly for diagnostic information. LUA communications require only the capabilities of PU 2.0; Host Integration Server or SNA Server provides these capabilities, plus the additional capabilities included in PU 2.1.
- The SSCP-LU session, between the LUA LU at the personal computer (PC) and the SSCP; this is used for controlling the LU.
- The LU-LU session, between the LUA LU at the PC and the host LU; this is used for data transfer between the PC and the host application.

LUA allows applications to send and receive data on the SSCP-LU session and on the LU-LU session. An LUA application can send data on this session using the common service verb TRANSFER_MS_DATA. LUA does not provide access to the PU-SSCP session.

The SSCP and LU sessions each provide two priorities of messages, normal and expedited. Expedited messages take precedence over other messages waiting to be transmitted on the same session. There are, therefore, four different flows on which a message can be sent or received:

- SSCP session (expedited flow)
- LU session (expedited flow)
- SSCP session (normal flow)
- LU session (normal flow)

The LU session normal flow carries most of the data; the other flows are used only for control purposes.

> **Note** The implementation of LUA in Host Integration Server or SNA Server does not allow applications to send data on the SSCP expedited flow and does not return data to an application on this flow.

Configuring for LUA

The Host Integration Server or SNA Server configuration file, which is set up and maintained by the system administrator, contains information that is required for LUA applications to communicate. An LUA LU is configured by the link service to use a connection to the host, and is given an LU number that matches that of an LU on the host.

The configuration can include LUA LU pools. A pool is a group of LUs with similar characteristics, and it allows an application to use any free LU from the pool. This feature can be used to allocate LUs on a first-come, first-served basis when there are more applications than LUs available, or to provide a choice of LUs on different connections.

The following communications components are configured for use with an LUA application.

Component	Description
Link service	A link service for communicating with the host. This component is normally configured by the setup program during Host Integration Server or SNA Server installation.
Connection	A connection to the host that uses the link service.
Local node	A local node that owns the connection. This component is configured automatically.
LUA LU	An LUA LU on the local node, configured to use the connection, with an LU number that matches an LU on the host.
LUA LU pool (optional)	If necessary, you can configure more than one LUA LU for the application, and group the LUs into a pool. This means that an application can specify the pool rather than a specific LU when issuing the RUI_INIT verb to start a session. Host Integration Server or SNA Server use this name in one of the following ways: If the name supplied is the name of an LU that is not in a pool, a session is assigned using that LU if it is available (that is, if it is not already in use by an LUA application). If the name supplied is the name of an LU pool, or the name of any LU within the pool, a session is assigned using the first available LU in the pool (if one is available). Note that this may not be the LU whose name was specified by the RUI_INIT verb.

LUA Verb Summary

LUA application programs can establish and use SNA sessions with either the RUI API or the SLI API. If an LUA application establishes an SNA session using **RUI_INIT**, it cannot issue any SLI verbs for that session. Likewise, if an LUA application establishes an SNA session using **SLI_OPEN**, it cannot issue any RUI verbs for that session.

Following is a brief summary of each LUA verb or user-supplied routine. Each verb supplies parameters to LUA, which performs the desired function and returns parameters to the application.

RUI_BID
: Allows the application to determine when information from the host is available to be read.

RUI_INIT
: Sets up the SSCP-LU session for an LUA application.

RUI_PURGE
: Cancels an outstanding **RUI_READ**.

RUI_READ
: Receives data or status information sent from the host to the LUA application's LU, on either the SSCP session or the LU session.

RUI_TERM
: Ends the SSCP session for an LUA application. It also terminates the LU session if it is active.

RUI_WRITE
: Sends data to the host on either the SSCP session or the LU session.

SLI_BID
: Notifies the SLI application that a message is waiting to be read using **SLI_RECEIVE**. It also provides the current status of the session to the LUA application.

SLI_BIND_ROUTINE
: An optional, user-supplied exit routine that notifies the LUA application that a BIND request has come from the host. It allows the routine to examine the request and formulate a response.

SLI_CLOSE
: Ends a session opened with **SLI_OPEN**.

SLI_OPEN
: Transfers control of the specified LU to the LUA application. It establishes a session between the SSCP and the specified LU, as well as an LU-LU session.

SLI_PURGE
: Cancels **SLI_RECEIVE** verbs issued with a wait condition.

SLI_RECEIVE
: Receives responses, SNA commands, and data into an LUA application's buffer. It also provides the current status of the session to the LUA application.

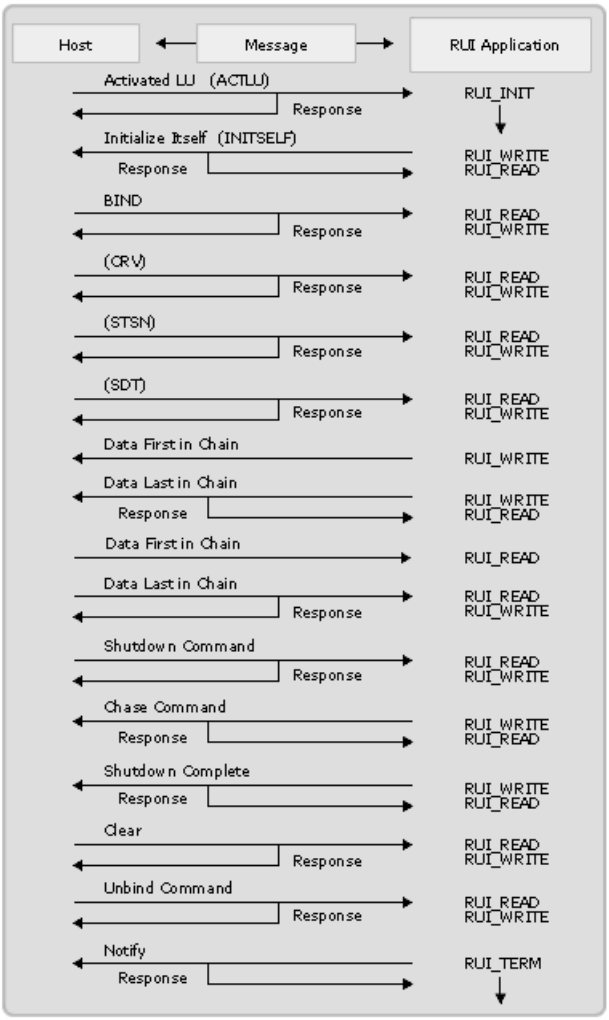
SLI_SEND
: Sends responses, SNA commands, and data from an LUA application to a host LU.

SLI_STSN_ROUTINE
: An optional, user-supplied exit routine that notifies the LUA application that a set and test sequence number (STSN) command has come from the host. It allows the routine to examine the request and formulate a response.

A Sample LUA Communication Sequence

This section illustrates how RUI and SLI verbs are used for an LUA communication sequence. The two figures illustrate the LUA verbs used to start a session, to exchange data, and to end the session, as well as the SNA messages sent and received. The arrows indicate the direction in which SNA messages flow.

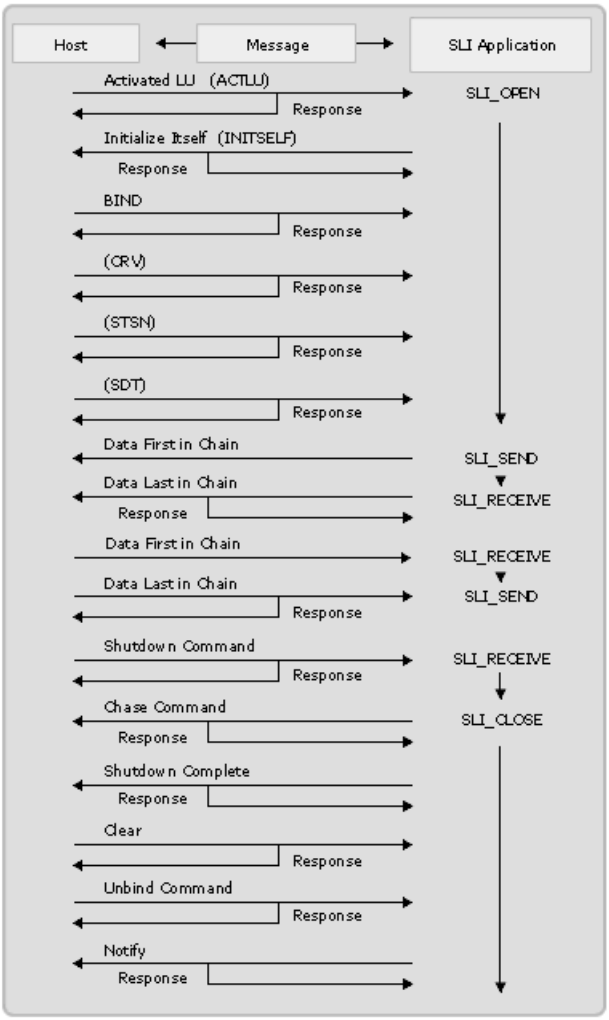
Communication Sequence Using RUI Verbs



In the example shown here, the application performs the following tasks:

- Issues an RUI_INIT verb to establish the SSCP session. (**RUI_INIT** does not complete until the LUA application has received an ACTLU message from the host and sent a positive response; however, these messages are handled by Host Integration Server or SNA Server and are not exposed to the LUA application.)
- Sends an INITSELF message to the SSCP to request a BIND and reads the response.
- Reads a BIND message from the host and writes the response. This establishes the LU session.
- Reads an SDT message from the host, which indicates that initialization is complete and data transfer can begin.
- Sends a chain of data consisting of three RUs (the last indicates that a definite response is required) and reads the response.
- Reads a chain of data consisting of three RUs and writes the response.
- Reads an UNBIND message from the host and writes the response. This terminates the LU session.
- Issues RUI_TERM to terminate the SSCP session. (Host Integration Server or SNA Server sends a NOTIFY message to the host and waits for a positive response; however, these messages are handled by Host Integration Server or SNA Server and are not exposed to the LUA application.)

Communication Sequence Using SLI Verbs



In the example shown here, the application performs the following tasks:

- Issues an SLI_OPEN verb to establish the SSCP session.
- Sends an INITSELF message to the SSCP to request a BIND and reads the response.
- Reads a BIND message from the host and writes the response. This establishes the LU session.
- Reads an SDT message from the host, which indicates that initialization is complete and data transfer can begin.

  > **Note** INITSELF, BIND, and SDT messages are handled by Host Integration Server or SNA Server if the application is using SLI. The **SLI_OPEN** does not return until Host Integration Server or SNA Server has sent an SDT and response.

- Issues SLI_SEND and SLI_RECEIVE to transfer data, SNA commands, or SNA responses between the host and the application.
- Issues SLI_CLOSE to terminate the SSCP session. (Host Integration Server or SNA Server sends a NOTIFY message to the host and waits for a positive response; however, these messages are handled by Host Integration Server or SNA Server and are not exposed to the LUA application.)

Writing LUA Applications

The information contained in this section will help you write LUA application programs for use with Microsoft® Host Integration Server or Microsoft SNA Server. The following topics are covered:

- LUA verbs and verb-control blocks
- Synchronous and asynchronous verb completion
- Compiling and linking an LUA application
- Resetting LUA LUs
- Multiple processes and multiple sessions
- Writing portable applications
- Programming techniques for LUA pools
- Microsoft® Windows® 2000, Microsoft Windows NT®, Microsoft Windows 98, Microsoft Windows 95, Microsoft Windows 3.x, Microsoft MS-DOS®, and OS/2 system considerations
- SNA considerations

Using LUA Verbs

This implementation of LUA is available to applications written in Microsoft® C version 5.1 or later. Applications access all LUA functions on Microsoft® Windows® 2000, Microsoft Windows NT®, Microsoft Windows 98, Microsoft Windows 95, and Microsoft Windows 3.x by issuing verbs using the external C functions RUI, SLI, WinRUI, and WinSLI.

Symbolic constants are defined in the WINLUA.H header file for many parameter values. Refer to the WINLUA.H file (contained in the Microsoft Host Integration Server or Microsoft SNA Server SDK) for a list of LUA constants.

You should use the symbolic constant and not the hexadecimal value when setting values for supplied parameters, or when testing values of returned parameters.

Parameters marked as reserved should always be set to zero.

RUI and SLI Definitions

The definitions of the RUI and SLI functions are as follows:

void WINAPI RUI(struct LUA_VERB_RECORD FAR * verb);

void WINAPI SLI(struct LUA_VERB_RECORD FAR * verb);

int WINAPI WinRUI(HWND handle, struct LUA_VERB_RECORD FAR * verb);

int WINAPI WinSLI(HWND handle, struct LUA_VERB_RECORD FAR * verb);

The WINLUA.H header file supplied with your Host Integration Server or SNA Server SDK includes prototypes of these functions.

The only parameter passed to the **RUI** or **SLI** function is the address of a verb control block (VCB). The VCB is a structure made up of variables that:

- Identify the LUA verb to be executed.
- Supply information used by the verb.
- Contain information returned by the verb when execution is complete.

The parameters passed to the WinRUI or WinSLI function are a window handle and the address of a VCB. The window handle is used for message notification when the issued verb has completed.

The VCB structure is declared in the WINLUA.H header file. See The LUA VCB Format for general VCB information. For verb-specific VCB information, see the reference documentation for each verb.

Issuing an LUA Verb

The following procedure is required to issue an LUA verb. In this example, the verb issued is RUI_INIT.

To issue an LUA verb

1. Create a variable for the VCB structure. For example:

```
#include <winlua.h>
        .
        .
struct LUA_VERB_RECORD rui_init;
```

2. The LUA_VERB_RECORD structure is declared in the WINLUA.H header file.
3. Clear (set to zero) the variables within the VCB:

```
memset( &rui_init, 0, sizeof( rui_init) );
```

   LUA requires that all reserved parameters, and all parameters not required by the verb being issued, must be set to zero. The simplest way to do this is to set the entire VCB to zeros before setting the parameters required for this particular verb.

4. Assign values to the VCB parameters that supply information to LUA:

```
rui_init.common.lua_verb = LUA_VERB_RUI;
rui_init.common.lua_verb_length = sizeof(struct LUA_COMMON);
rui_init.common.lua_opcode = LUA_OPCODE_RUI_INIT;
memcpy (rui_init.common.lua_luname, "THISLU  ", 8);
```

   The values LUA_VERB_RUI and LUA_OPCODE_RUI_INIT are symbolic constants. These constants are defined in the WINLUA.H header file in the SNA Server SDK. To ensure portability between different systems, use symbolic constants and not integer values.

5. Invoke LUA. The only parameter is a pointer to the address of the structure containing the VCB for the desired verb.

```
RUI( &rui_init );
```

6. Check the asynchronous flag (**rui_init.common.lua_flag2.async**) to determine whether the verb completed asynchronously. If events are being used and the verb did complete asynchronously, wait for the event to complete.

```
if (rui_init.common.lua_flag2.async)
{
/* verb will complete asynchronously so continue
with other processing */
/* then wait */
WaitForSingleObject (...)
}
```

   Do not check the return code; it may have changed from LUA_IN_PROGRESS to LUA_OK by the time you check it.

7. Check the variables returned by LUA.

```
if( rui_init.common.lua_prim_rc == LUA_OK )
{
/* Init OK */
        .
```

```
        .
}
else
{
/* Do error routine */
        .
        .
}
```

The LUA VCB Format

The LUA verb control block (VCB) is called LUA_VERB_RECORD. It is a structure with two parts:

- A structure, LUA_COMMON, which is used for all the LUA verbs.
- A union, LUA_SPECIFIC, which is used only by RUI_BID, SLI_BID, SLI_OPEN, and SLI_SEND.

The following topics describe the VCB and its parts.

LUA_VERB_RECORD

The LUA VCB structure is as follows:

typedef struct LUA_VERB_RECORD {

struct LUA_COMMON common;

union LUA_SPECIFIC specific;

} LUA_VERB_RECORD;

To access parameters in the common part of the VCB, you need to include the structure member name **common**. For example, when using a verb record structure named **Lua_Verb**, you access its **lua_prim_rc** member as **Lua_Verb.common.lua_prim_rc**.

To access parameters in the specific part of the VCB, you need to include the union member name **specific.** For example, when issuing RUI_BID using a verb record structure named **Lua_Verb**, you access its **lua_peek_data** member as **Lua_Verb.specific.lua_peek_data**.

For a complete listing of the structures and related values in the LUA VCB, see LUA Verb Control Blocks.

LUA_COMMON

The following structure lists the common data structure parameters used by all the LUA verbs.

```
struct LUA_COMMON {
unsigned short lua_verb;
unsigned short lua_verb_length;
unsigned short lua_prim_rc;
unsigned long lua_sec_rc;
unsigned short lua_opcode;
unsigned long lua_correlator;
unsigned char lua_luname[8];
unsigned short lua_extension_list_offset;
unsigned short lua_cobol_offset;
unsigned long lua_sid;
unsigned short lua_max_length;
unsigned short lua_data_length;
char FAR * lua_data_ptr;
unsigned long lua_post_handle;
struct LUA_TH lua_th;
struct LUA_RH lua_rh;
struct LUA_FLAG1 lua_flag1;
unsigned char lua_message_type;
struct LUA_FLAG2 lua_flag2;
unsigned char lua_resv56[7];
unsigned char lua_encr_decr_option;
} LUA_COMMON;
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs or LUA_VERB_SLI for SLI verbs. For both of these macros the value is 0x5200.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, for example, LUA_OPCODE_RUI_BID for the RUI_BID verb.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_OPEN and RUI_INIT require this parameter. Other Windows LUA verbs only require this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Specifies the offset from the start of the VCB to the extension list of user-supplied dynamic-link libraries (DLLs). Not used by RUI in Host Integration Server or SNA Server and should be set to zero.

lua_cobol_offset
Offset of the Cobol extension. Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Specifies the length of received buffer for RUI_READ and SLI_RECEIVE. For other RUI and SLI verbs, it is not used and should be

set to zero.

lua_data_length

Returned parameter. Specifies the length of data returned in **lua_peek_data** for the RUI_BID verb.

lua_data_ptr

Pointer to the application-supplied buffer that contains the data to be sent for SLI_SEND and RUI_WRITE or that will receive data for **SLI_RECEIVE** and **RUI_READ**. For other RUI and SLI verbs, this parameter is not used and should be set to zero.

lua_post_handle

Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th

Returned parameter. Contains the SNA transmission header (TH) of the message sent or received. Various subparameters are set for write functions and returned for read and bid functions.

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits.

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh

Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for the write function and returned by the read and bid functions.

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits.

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. This parameter is used by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, SLI_RECEIVE, and SLI_SEND. For other LUA verbs this parameter is not used and should be set to zero.

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. This is a returned parameter for RUI_INIT and

SLI_OPEN and a supplied parameter for SLI_SEND. For other LUA verbs this variable is not used and should be set to zero.

Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

The SLI receives and responds to the BIND, CRV, and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. This parameter is returned by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, SLI_RECEIVE, and SLI_SEND. For other LUA verbs this parameter is not used and should be set to zero.

lua_flag2.bid_enable

Indicates that **RUI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Supplied parameter. Reserved field used by SLI_OPEN and RUI_INIT. For all other LUA verbs, this parameter is reserved and should be set to zero.

lua_encr_decr_option
Field for cryptography options. On **RUI_INIT**, only the following are supported:

- **lua_encr_decr_option** = 0
- **lua_encr_decr_option** = 128

For all other LUA verbs, this parameter is reserved and should be set to zero.

LUA_SPECIFIC

The following union shows the specific data structure that is included for functions that use the **LUA_SPECIFIC** part of a verb control block. The only LUA verbs that use this union are RUI_BID, SLI_BID, SLI_OPEN, and SLI_SEND.

```
union LUA_SPECIFIC {
struct SLI_OPEN open;
unsigned char lua_sequence_number[2];
unsigned char lua_peek_data[12];
} LUA_SPECIFIC;
```

Members

open
The union member of **LUA_SPECIFIC** used by the **SLI_OPEN** verb.

lua_sequence_number
The union member of **LUA_SPECIFIC** used by the **SLI_SEND** verb. Returned parameter. Sequence number of the RU to the host.

lua_peek_data
The union member of **LUA_SPECIFIC** used by the **RUI_BID** and **SLI_BID** verbs. Returned parameter. Contains up to 12 bytes of the data waiting to be read.

LUA_SPECIFIC.SLI_OPEN

The following structure shows the **SLI_OPEN** fields of the **LUA SPECIFIC** union member for the SLI_OPEN verb.

```
struct SLI_OPEN {
unsigned char lua_init_type;
unsigned char lua_resv65;
unsigned short lua_wait;
struct LUA_EXT_ENTRY lua_open_extension[3];
unsigned char lua_ending_delim;
} SLI_OPEN;
```

Members

lua_init_type
Type of session initiation.

lua_resv65
Reserved field.

lua_wait
Secondary retry wait time.

lua_open_extension
Supplied parameter. Specifies any user-supplied dynamic-link libraries (DLLs) used to process specific LUA messages.

lua_ending_delim
Extension list delimiter.

LUA_EXT_ENTRY

The following structure shows the **LUA_EXT_ENTRY** fields of the LUA SPECIFIC.SLI_OPEN union member for the SLI_OPEN verb.

```
struct LUA_EXT_ENTRY {
unsigned char lua_routine_type;
unsigned char lua_module_name[9];
unsigned char lua_procedure_name[33];
};
```

Members

lua_routine_type
Extension routine type.

lua_module_name
Extension DLL module name.

lua_procedure_name
Procedure name to call in the extension DLL module.

LUA Synchronous and Asynchronous Verb Completion

LUA verbs can complete execution either synchronously or asynchronously.

Synchronous Verb Completion

When LUA is able to complete all the processing for a verb as soon as it is issued, the verb has completed synchronously. When this happens, the primary return code is set to a value other than LUA_IN_PROGRESS, and the **lua_flag2.async** bit is set to zero. Note that the value of the **lua_flag2.async** bit should be tested, not the primary return code being not equal to LUA_IN_PROGRESS. (See individual verb descriptions for information on these returned parameters.)

Asynchronous Verb Completion

Some LUA verbs (for example, RUI_PURGE) complete quickly after local processing; however, most verbs take some time to complete because they require messages to be sent to and received from the local node or the host application.

When LUA must wait for information from the remote LU or from the local node before it can complete a verb, the verb completes asynchronously.

When this happens, the **lua_flag2.async** bit is set to 1. The primary return code is also normally set to LUA_IN_PROGRESS, but this value cannot be relied on. The value of the **lua_flag2.async** bit should be tested. The application can now perform other processing, or wait for notification from LUA that the verb has completed. LUA issues this notification by setting the primary return code to its final value and leaving **lua_flag2.async** set to 1.

When the verb completes, LUA does the following depending on your environment:

- For a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system, two types of notification are possible. The LUA application either:

  Issues **WaitForSingleObject** or **WaitForMultipleObject**.

  —or—

  Posts the "WinRUI"/"WinSLI" notification message to the window handle of the **WinRUI**/**WinSLI** message.

  The event method using **WaitForSingleObject** or **WaitForMultipleObject** is the preferred way to receive asynchronous notification on Windows 2000, Windows NT, Windows 98, or Windows 95.

- In the Windows environment, it notifies the completion of an asynchronous request by posting the "WinRUI"/"WinSLI" notification message to the window handle of the **WinRUI**/**WinSLI** message. A window handle has been added as the first parameter passed to the WinRUI and WinSLI entry points.
- For OS/2, it notifies the completion of an asynchronous request by posting a message to the application's Window procedure or by clearing a semaphore/event. Additionally, the application can use system queues for asynchronous verbs.

  Note that if the verb completes synchronously, LUA does not clear the semaphore.

Compiling and Linking an LUA Application

Use the following procedure to compile and link a LUA application.

To compile and link an LUA application for use with Host Integration Server or SNA Server

1. Update the **path** statement to include the directory containing the LUA application.
2. Set any required environmental variables.
3. Compile the application, including the WINLUA.H header file provided in the Microsoft® Host Integration Server or Microsoft® SNA Server SDK, to produce the .OBJ files.
4. Link the application with the WINLUA.LIB library to produce an .EXE file.

Resetting LUA LUs

Both Microsoft® Host Integration Server and Microsoft® SNA Server provide a facility for resetting LUA LUs or forcing off LUA applications, which is useful if an application has become deadlocked or is looping.

The NetView command **deactivate-oldlu** can also be used to reset an LUA LU. These facilities interact with the LUA application as described in the following paragraphs.

When an LUA LU is reset through Host Integration Server or SNA Server or by using the **deactivate-oldlu** command, Host Integration Server or SNA Server sends an UNBIND message to the application (as though the host had issued it).

The UNBIND message sent to the application is 0x32 0x0E, indicating a recoverable LU failure, and is returned to the application on a subsequent RUI_READ. The LU session is terminated, but the SSCP session remains active (that is, the LU is returned to the same state as if RUI_INIT has just completed).

Multiple Processes and Multiple Sessions Using LUA

Two processes cannot use the same LUA session. Only the process that issues RUI_INIT can use the session that is started by the verb. Before another process can use LUA, it must issue **RUI_INIT** to obtain a new session. However, different threads of the same process can issue verbs for the same LUA session.

A single process can simultaneously use more than one LUA session by issuing multiple **RUI_INIT** verbs. Win32 processes support for up to 15,000 sessions for applications based on the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95. Processes on OS/2 systems support up to 512 sessions. Processes on Microsoft® Windows 3.x support up to 16 sessions. Each session must use a different LU. Two or more sessions can use the same pool, but the **lua_luname** member (which is either the name of the pool or the name of an LU within the pool) must be different for each **RUI_INIT**.

Two or more instances of the same LUA application can run as different processes, but they must use different LUs. This can be done by using LU pools; the two processes can specify the same pool, but are allocated different LUs from that pool.

Programming Techniques for LUA Pools

When working with an LUA LU pool, specify the pool name at the beginning of the conversation and then use the **lua_sid** member (not the pool name) with subsequent calls. This is necessary because the **lua_sid** member is a unique identifier, but the pool name is not, because a pool is designed to supply LUs for multiple conversations.

When using RUI_INIT or SLI_OPEN with an LUA pool, specify the pool name with the **lua_luname** member. For subsequent calls in the same conversation, use the **lua_sid** member returned from **RUI_INIT** or **SLI_OPEN** to specify the conversation.

Also note that on completion of **RUI_INIT** or **SLI_OPEN**, the **lua_luname** member contains the actual name of the LU used. This allows you to create code for a display for the user, showing the actual LU name used in a particular conversation.

Writing Portable LUA Applications

Use the following guidelines for writing LUA applications that are portable to other environments:

- the symbolic constant names for parameter values and return codes, not the numeric values shown in the WINLUA.H file. (See the WINLUA.H file in the Microsoft® Host Integration Server or Microsoft® SNA Server SDK for more information.)
- When accessing SNA sense codes in a data buffer, use the symbolic constants rather than the numeric values; this ensures that the byte storage order is correct for your particular system. You should use **memcpy** to set the values, and **memcmp** to test them. For example:

```
memcpy (this_verb.common.lua_data_ptr, LUA_INCORRECT_REQ_CODE, 4);

if (memcmp (this_verb.common.lua_data_ptr,
LUA_INCORRECT_REQ_CODE, 4) == 0)
{
.....
}
```

- Ensure that any parameters shown as reserved are set to zero.
- Set the **lua_verb_length** parameter as described in the verb description.

LUA System Considerations

This section provides specific information about developing LUA applications for the following operating systems:

- LUA Considerations on Microsoft Windows 2000, Windows NT, Windows 98, and Windows 95
- LUA Considerations on Microsoft Windows 3.x
- LUA Considerations on Microsoft MS-DOS
- LUA Considerations on OS/2

LUA Considerations on Microsoft Windows 2000, Windows NT, Windows 98, and Windows 95

This section summarizes information for developing Win32® LUA applications for Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95.

Byte ordering

The values of constants defined in the WINLUA.H file are dependent on the byte ordering of the hardware used. Macros are used to set the constants to the correct value.

Currently, the include files in the Windows 2000 or Windows NT system use the i386 macro (Windows NT also uses the ALPHA, MIPS, and PPC macros) to indicate the hardware. These same macros are used by Microsoft® Host Integration Server or Microsoft® SNA Server, along with the Win32 macro, to indicate the byte ordering needs. The macros must be defined in the application or on the command line when building the application.

For example, the primary return code of LUA_PARAMETER_CHECK is defined to have a value of 0x0001. Depending on the environment, the constant LUA_PARAMETER_CHECK may or may not be 0x0001. Some formats define the value as it appears in memory; others define it as a 2-byte variable. Because it cannot be assumed that an application will always use provided constants rather than hardwired values, a macro can be defined to swap the bytes. The following example shows how the macro can be used:

```
#define LUA_PARAMETER_CHECK LUA_FLIPI (0X0001)
```

Events

To receive data asynchronously, an event handle is passed in the semaphore field of the VCB. This event must be in the nonsignaled state when passed to LUA, and the handle must have EVENT_MODIFY_STATE access to the event.

Library names

To support the coexistence of Win16 and Win32 API libraries on the same computer, the Win32 DLL names have been changed from the names used by Win16 API libraries. Win32 stub DLL libraries using the old names are supplied with Host Integration Server or SNA Server so that older applications are still supported.

Old DLL names	New DLL names
WINRUI.DLL	WINRUI32.DLL
WINSLI.DLL	WINSLI32.DLL

The old DLL names should be used for Win32-based applications that are required to run on SNA Server version 2.0. The new DLL names should be used for Win32-based applications that are intended to run only on Host Integration Server or on SNA Server version 2.1 or later.

If you intend your Win32-based application to be used with SNA Server version 2.0, you should link with the libraries included with SNA Server version 2.0. Otherwise, use the libraries provided with Host Integration Server or with SNA Server version 2.1 or later.

Load-time linking

To be dynamically linked to LUA at load time, you must do one of the following at link time:

- Insert the following **IMPORTS** statements in the definition (.DEF) file used to link the application:

  (For RUI)

  ```
  IMPORTS WINRUI.RUI
  IMPORTS WINRUI WinRUI
  IMPORTS WINRUI.WinRUIStartup
  IMPORTS WINRUI.WinRUICleanup
  ```

  (For SLI)

  ```
  IMPORTS WINSLI.SLI
  IMPORTS WINSLI.WinSLI
  IMPORTS WINSLI.WinSLIStartup
  ```

```
IMPORTS WINSLI.WinSLICleanup
```

- Link the application to WINRUI.LIB (for RUI) or WINSLI.LIB (for SLI), which contain the entry-point linkage information.

Multiple threads

An LUA application can have multiple threads that issue verbs. LUA for the Win32 system makes provisions for multithreading processes on Windows 2000, Windows NT, Windows 98, and Windows 95. A process contains one or more threads of execution. All references to threads refer to actual threads in a multithreaded Windows 2000, Windows NT, Windows 98, or Windows 95 environment.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or DWORD boundaries, whichever is smaller. As a result, DWORDs are aligned on DWORD boundaries, WORDs are aligned on WORD boundaries, and BYTEs are aligned on BYTE boundaries. This means, for example, that there is a 2-byte gap between the primary and secondary return codes. Therefore, the elements in a VCB should only be accessed using the structures provided.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Registering and deregistering applications

All LUA applications for the Windows 2000, Windows NT, Windows 98, or Windows 95 system must call the Windows SNA extension WinRUIStartup or WinSLIStartup at the beginning of the session to register the application and WinRUICleanup or WinSLICleanup at the end of the session to deregister the application.

Restrictions on 3270-style LUs

A Windows 2000, Windows NT, Windows 98, or Windows 95 process cannot access 3270-style LUs from both the Function Management Interface (FMI) and LUA APIs at the same time. However, the process can use the LUA APIs to access LUA LUs while using FMI APIs to access 3270-style LUs.

Run-time linking

For an application to be dynamically linked to LUA at run time, it must issue the following calls:

- **LoadLibrary** to load the specified library module for Windows LUA. That is, WINRUI.DLL or WINRUI32.DLL (for RUI), and WINSLI.DLL or WINSLI32.DLL (for SLI).
- **GetProcAddress** to retrieve the address of the LUA function entry points exported by the DLL. For RUI, the function entry points are RUI, WinRUI, WinRUIStartup, and WinRUICleanup. For SLI, the function entry points are SLI, WinSLI, WinSLIStartup, and WinSLICleanup.

LUA Considerations on Microsoft Windows 3.x

This section summarizes information for developing LUA applications on a Microsoft® Windows® version 3.*x* system.

Load-time linking

To be dynamically linked to LUA at load time, you must do one of the following at link time:

- Insert the following **IMPORTS** statements in the definition (.DEF) file used to link the application:

  (For RUI)

  ```
  IMPORTS WINRUI.RUI
  IMPORTS WINRUI WinRUI
  IMPORTS WINRUI.WinRUIStartup
  IMPORTS WINRUI.WinRUICleanup
  ```

  (For SLI)

  ```
  IMPORTS WINSLI.SLI
  IMPORTS WINSLI.WinSLI
  IMPORTS WINSLI.WinSLIStartup
  IMPORTS WINSLI.WinSLICleanup
  ```

- Link the application to WINRUI.LIB (for RUI) or WINSLI.LIB (for SLI), which contain the entry-point linkage information.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDs are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Registering and deregistering applications

All Windows LUA applications must call the Windows SNA extension function WinRUIStartup or WinSLIStartup at the beginning of the session to register the application and WinRUICleanup or WinSLICleanup at the end of the session to deregister the application.

Run-time linking

For an application to be dynamically linked to LUA at run time, it must issue the following calls:

- Call the **LoadLibrary** function to dynamically load the WINRUI.DLL (for RUI) or WINSLI.DLL (for SLI) library module for Windows LUA.
- Call the **GetProcAddress** function to retrieve the address of each LUA function entry point that will be called in the DLL. For RUI, the function entry points are RUI, WinRUI, WinRUIStartup, and WinRUICleanup. For SLI, the function entry points are SLI, WinSLI, WinSLIStartup, and WinSLICleanup.

Simultaneous sessions

An application can participate in as many as 16 sessions simultaneously in the Windows environment. However, if multiple LUA applications are active at the same time, the total number of sessions cannot exceed 16.

LUA Considerations on Microsoft MS-DOS

This section summarizes information for developing LUA applications on a Microsoft® MS-DOS® system. Only RUI is supported on MS-DOS; SLI is not supported on MS-DOS.

Implementing semaphores

RUI signals that an asynchronous verb has completed by setting **lua_post_handle** to zero. For example:

```
/*Set lua_post_handle to non-zero value */

rui_innit.common.lua_post_handle = 1l ;

/*Issue RUI verb */

RUI (&rui_init) ;

/*Check to see if verb will complete asynchronously */

if (rui_init.common.lua_flag2.async)

{

/*Verb will complete asynchronously */
```

To wait for the verb to complete, loop until **lua_post_handle** is cleared. For example:

```
while (rui_init.common.lua_post_handle != 0)

{

/* The application is free to do other work in this loop */

}

/* The loop has exited, so the verb completed */
```

Keep in mind the following guidelines:

- The application should set **lua_post_handle** to zero before it issues a verb.
- A single MS-DOS application can run up to 16 sessions, but memory limitations can cause problems if the application sends anything except very small RUs. A more realistic limit is two to three sessions per application.

Load-time linking

To be dynamically linked to LUA at load time, you must link the application to DOSACS.LIB, which contains the entry-point linkage information.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDs are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

LUA Considerations on OS/2

This section summarizes information for developing LUA applications on OS/2.

This implementation of LUA is binary-compatible with the implementation of the RUI and SLI interfaces in IBM Extended Services (ES) for OS/2 version 1.0 LUA. Therefore, there are no migration steps for IBM ES for OS/2 version 1.0 applications. However, if you recompile an IBM ES for OS/2 version 1.0 LUA application for use with LUA, include the WINLUA.H header file provided in the SNA Server SDK to ensure complete platform compatibility.

Note that this LUA implementation does not support user-defined encryption and decryption routines.

Critical sections

Exercise great caution when using critical sections, which are the parts of a program that must run without interruption. An application must not issue an LUA verb within a critical section.

Data segments

Data is sent from and received in data buffers established by the application. A data buffer must reside on an unnamed shared data segment and must fit entirely within the data segment. Many data buffers can reside on the same data segment.

The data segment for the VCB must have read and write attributes. It can be one of the following:

- A variable (not a local variable because LUA will copy data to it on completion of the verb).
- Allocated dynamically using **DosAllocSeg** or **DosSubAlloc**.

Load-time linking

For an application to be dynamically linked to LUA at load time, you must do one of the following at link time:

- Insert the following **IMPORTS** statement in the definition (.DEF) file used to link the application:

  (For RUI)

  ```
  IMPORTS RUI.RUI
  ```

  (For SLI)

  ```
  IMPORTS SLI.SLI
  ```

- Link the application to ACSRUI.LIB (for RUI) or ACSSLI.LIB (for SLI), which contain the entry-point linkage information for various APIs.

Multiple threads

An application session can have multiple threads that issue verbs. However, the same thread of an application cannot issue two verbs simultaneously. If LUA is executing a verb and the same thread of the application issues a verb, LUA returns the LUA_UNSUCCESSFUL return code to the later verb and leaves it unexecuted.

OS/2 exception TRAP 000D

The OS/2 exception TRAP 000D is issued when LUA is unable to pass a return code to the application for one of the following reasons:

- The data segment containing the VCB is not read/writable.
- The VCB is fewer than 10 bytes in length.
- The semaphore supplied through the **lua_post_handle** parameter is neither a valid RAM or OS/2 system semaphore nor a pointer to a location within a writable segment.

Packing

For performance considerations, the VCBs are not packed. VCB structure member elements after the first element are aligned on either the size of the member type or WORD (2-byte) boundaries, whichever is smaller. As a result, DWORDs and WORDS are aligned on WORD boundaries and BYTEs are aligned on BYTE boundaries. For portability to Win32, VCBs should be accessed using the structures provided since the alignment of structure members differs.

This option for structure and union member alignment is the default behavior for Microsoft C/C++ compilers producing 16-bit code. For compatibility with the supplied LUA libraries, make sure to use an equivalent structure and union member packing

option when using other C/C++ compilers or when explicitly specifying a structure alignment option when using Microsoft compilers.

Process support

A single RUI or SLI LU can be used by only a single process. A single process can act as multiple RUI/SLI applications and have sessions using multiple LUs.

Run-time linking

For an application to be dynamically linked to LUA at run time, it must issue the following calls:

- Call the **DosLoadModule** function to dynamically load the ACSRUI.DLL (for RUI) or ACSSLI.DLL (for SLI) library module for LUA.
- Call the **DosGetProcAddr** function to set the entry point **RUI** or **SLI** to the dynamic-link library.

Unlinking (the **DosFreeModule** call) is not supported.

Simultaneous sessions

On Windows 2000, Windows NT, Windows 98, and Windows 95, an application can participate in as many as 15,000 sessions simultaneously. In the Windows 3.x environment, an application can participate in as many as 16 sessions simultaneously. In OS/2, an application can participate in as many as 512 sessions simultaneously.

Stack size

The recommended stack size for an application is at least 3000 bytes.

When executing a verb, LUA uses the calling application's stack. The combination of OS/2 and LUA requires 2560 bytes of stack space, and the application requires additional stack space for its variables.

VCB segment

The segment containing the VCB must be a writable segment. All reserved and unused fields in the VCB should be set to 0x00.

SNA Considerations Using LUA

This section explains SNA information you need to consider when writing LUA applications.

BIND checking

During initialization of the LU session, the host sends to the LUA application a BIND message that contains information such as RU sizes for use by the LU session. Microsoft® Host Integration Server or Microsoft® SNA Server returns this message to the LUA application on RUI_READ. The LUA application must verify that the parameters specified on the BIND are suitable. The application has the following options:

- It can accept the BIND as it is, by issuing RUI_WRITE containing an OK response to the BIND. No additional BIND data can be sent on the response.
- It can try to negotiate one or more BIND parameters (this is only permitted if the BIND is negotiable). To do this, the application issues **RUI_WRITE** containing an OK response, but including the modified BIND as data.
- It can reject the BIND by issuing **RUI_WRITE** containing a negative response, using an appropriate SNA sense code as data.

Note that validating the BIND parameters, and ensuring that all messages sent are consistent with them, is the responsibility of the LUA application. However, the following two restrictions apply:

- Host Integration Server or SNA Server rejects any **RUI_WRITE** that specifies an RU length greater than the size specified on the BIND.
- Host Integration Server or SNA Server requires the BIND to specify that the secondary LU is the contention winner and that error recovery is the responsibility of the contention loser.

Note For SLI, an application must specify that it will use SLI_BIND_ROUTINE on the SLI_OPEN if it will do any BIND checking.

Courtesy acknowledgments

Host Integration Server or SNA Server keeps a record of requests received from the host in order to correlate any response sent by the application with the appropriate request. When the application sends a response, Host Integration Server or SNA Server correlates the response with the data from the original request, and can then free the storage associated with it.

If the host specifies exception response only (a negative response can be sent but a positive response should not be sent), Host Integration Server or SNA Server must still keep a record of the request in case the application subsequently sends a negative response. If the application does not send a response, the storage associated with this request cannot be freed.

Because of this, Host Integration Server or SNA Server allows the LUA application to issue a positive response to an exception-response-only request from the host (this is known as a courtesy acknowledgment). The response is not sent to the host, but is used by LUA to clear the storage associated with the request.

Note that the application does not need to send a courtesy acknowledgment for each exception-response-only request. For efficiency, the application can respond less frequently. The node treats a courtesy acknowledgment as an implicit acknowledgment for all prior pending requests.

Distinguishing SNA sense codes from other secondary return codes

A secondary return code that is not a sense code always contains a value of zero in its first two bytes.

An SNA sense code always contains a nonzero value in its first two bytes; the first byte gives the sense code category and the second identifies a particular sense code within that category. (The third and fourth bytes can contain additional information or can be zero.)

Information on SNA sense codes

If you need information on a returned sense code, see Sense Codes in the SNA Formats document. The sense codes are listed in numerical order by category.

Negative responses and SNA sense codes

SNA sense codes can be returned to an LUA application in the following cases:

- When the host sends a negative response to a request from the LUA application, it includes an SNA sense code indicating the reason for the negative response. This is reported to the application on a subsequent RUI_READ or SLI_RECEIVE with the following information:

Primary return code	LUA_OK
Request/response indicator, response type indicator, and sense data included indicator	All set to 1, indicating a negative response that includes sense data.

Data returned	The SNA sense code.

- When Host Integration Server or SNA Server receives invalid data from the host, it generally sends a negative response to the host and does not pass the invalid data to the LUA application. This is reported to the application on a subsequent RUI_READ, SLI_RECEIVE, RUI_BID, or SLI_BID with the following information:

Primary return code	LUA_NEGATIVE_RESPONSE
Secondary return code	The SNA sense code sent to the host.

- In some cases, Host Integration Server or SNA Server detects that data supplied by the host is invalid, but cannot determine the correct sense code to send. In this case, it passes the invalid data in an exception request (EXR) to the LUA application on RUI_READ or SLI_RECEIVE with the following information:

Request/response indicator	Set to 0, indicating a request.
Sense data included indicator	Set to 1, indicating that sense data is included. (This indicator is normally used only for a response.)
Message data	A suggested SNA sense code.

  The application must then send a negative response to the message; it can use the sense code suggested by Host Integration Server or SNA Server, or it can alter the sense code.

- Host Integration Server or SNA Server can send a sense code to the application to indicate that data supplied by the application was invalid. This is reported to the application on RUI_WRITE or SLI_SEND with the following information:

Primary return code	LUA_UNSUCCESSFUL
Secondary return code	SNA sense code.

  The sense codes that can be returned as secondary return codes on LUA verbs are listed in the WINLUA.H header file; see the Host Integration Server or SNA SDK for this file.

Pacing

Pacing is handled by the LUA interface; an LUA application does not need to control pacing and should never set the pacing indicator flag.

If pacing is being used on data sent from the LUA application to the host (this is determined by the BIND), RUI_WRITE or SLI_SEND may take some time to complete. This is because LUA has to wait for a pacing response from the host before it can send more data.

If an LUA application transfers large quantities of data in one direction, either to the host or from the host (for example, a file transfer application), then the host configuration should specify that pacing is used in that direction. This ensures that the node receiving the data is not flooded with data and does not run out of data storage.

Purging data to end of chain

When the host sends a chain of request units to an LUA application, the application can wait until the last RU in the chain is received before sending a response, or it can send a negative response to an RU that is not the last in the chain. If a negative response is sent midchain, LUA purges all subsequent RUs from this chain and does not send them to the application.

When LUA receives the last RU in the chain, it indicates this to the application by setting the primary return code of RUI_READ or RUI_BID to LUA_NEGATIVE_RESPONSE with a zero secondary return code.

Note that the host can terminate the chain by sending a message such as CANCEL while in midchain. In this case, the CANCEL message is returned to the application on **RUI_READ**. The LUA_NEGATIVE_RESPONSE return code is not used.

Segmentation

Segmentation of RUs is handled by the LUA interface. LUA always passes complete RUs to the application, and the application should pass complete RUs to LUA.

Support for LUA Single Sign-On

This section describes the LUA application support for single sign-on using 3270 display sessions that is available in Microsoft® Host Integration Server 2000 and in Microsoft® SNA Server version 3.0 with Service Pack 1 or higher.

Over 3270 LUs, a single sign-on feature for LUA applications is supported to automate the overall logon process. When configured for this feature, Host Integration Server or SNA Server automatically replaces special keywords in the data stream with the actual host user name and password at appropriate points in the session.

Note that single sign-on is not supported over LUA LUs.

To open 3270 LUs from an LUA application using RUI, the lua_resv56[1] field must be set to a non-zero value when this verb control block is passed to RUI_INIT. To open 3270 LUs from an LUA application using SLI, the lua_resv56[2] field must be set to a non-zero value when this verb control block is passed to SLI_OPEN. Please see the reference sections on **RUI_INIT** and **SLI_OPEN** for details.

Prerequisites for LUA Single Sign-On

In preparation for using LUA single sign-on over 3270 LUs, the system administrator must define a host security domain containing host connections. This host security domain must be initially created or modified to enable the single sign-on feature. The system administrator must enable a user's Microsoft® Windows 2000 or Microsoft® Windows NT® account in the host security domain and either the administrator or the user must establish a mapped host account for the Windows 2000 or Windows NT domain user name.

The user must be logged on to a Windows 2000 or Windows NT domain with a user name and password. Note that this single sign-on feature is only supported over 3270 LUs.

Registry Settings Used for LUA Single Sign-On

The LUA single sign-on feature depends on Host Integration Server or SNA Server scanning 3270 LUs used in the logon process for special keywords that are defined in the registry on the computer running Host Integration Server or SNA Server. The values for these special keywords can be defined by the system administrator on the computer running Host Integration Server or SNA Server.

The registry settings used by the LUA single sign-on process are located under the **HKEY_LOCAL_MACHINE\CurrentControlSet\Services** registry node. Installed under the **SNASERVR\PARAMETERS** subkey are the following entries:

3270SSOPadByte
This entry should be set to an ASCIIZ string to use as the character for padding replacement text in the user name or password if these strings are shorter than the length of the special tag strings defined below. The default value for this pad character is the ASCII space character.

3270SSOPostReplaceCount
This entry should be set to a DWORD that represents the number of message chains of RUs to scan after replacement of text for user name or password. The default value for this number is 10.

3270SSOPrefix
This entry should be set to an ASCIIZ string to use as the special prefix tag string in combination with the user name and password tags. The default value of this string is MS$.

3270SSOPwdTag
This entry should be set to an ASCIIZ string to use as the special tag string in combination with the **3270SSOPrefix** tag in defining the special host password string that will be replaced. The default value of this string is SAMEP, so the default host password string that is scanned for and replaced is MS$SAMEP. Note that length of the password string that is scanned for (MS$SAMEP, for example) determines the maximum length of the password string that can sent to the host using single sign-on. This limit occurs because the password substitution cannot change the length of the data message
Note that the value of this string must be different from the value of the **3270SSOUserTag** entry for single sign-on to function properly.

3270SSOReplaceCount
A DWORD value that affects the timeout value for password substitution. User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the timer expires. By default the timer will be set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node.

If the **3270SSOReplaceCount** registry entry is defined and the **3270SSOReplaceTimer** registry entry is not defined, the node counts this number of RUs (on PLU-SLU session only) before timeout occurs. If both the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries are defined, the value for **3270SSOReplaceCount** will be used to determine when a timeout will occur. By default, this key is not defined and the node defaults to a timeout of 30 seconds.

3270SSOReplaceTimer
A DWORD value that affects the timeout value for password substitution. User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the timer expires. By default the timer will be set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node.

If the **3270SSOReplaceTimer** registry entry is defined and **3270SSOReplaceCount** is not defined, the node uses this value in seconds before timeout occurs. If both the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries are defined, the value for **3270SSOReplaceCount** will be used to determine when a timeout will occur. By default, this key is not defined and the node defaults to a timeout of 30 seconds.

3270SSOUserTag
This entry should be set to an ASCIIZ string to use as the special tag string in combination with the **3270SSOPrefix** tag in defining the special user name string that will be replaced. The default value of this string is SAMEU, so the default user name string that is scanned for and replaced is MS$SAMEU. Note that length of the user name string that is scanned for (MS$SAMEU, for example) determines the maximum length of the user name string that can sent to the host using single sign-on. This limit occurs because the user name substitution cannot change the length of the data message
Note that the value of this string must be different from the value of the **3270SSOPwdTag** entry for single sign-on to function properly.

LUA User Name and Password Replacement

The SNA node on the host monitors the inbound session for a replacement sequence consisting of the **3270SSOPrefix** string immediately followed by one of the strings **3270SSOUserTag** or **3270SSOPwdTag**. Thus, the default user name string that would be scanned for and replaced is MS$SAMEU. When this string is found in the inbound session data, the node looks up the corresponding information (host user name in the current host security domain) and overwrites MS$SAMEU with this information. The same process occurs for the password string that would be scanned for and replaced, which defaults to MS$SAMEP.

Note that this operation cannot change the length of the data message. If the actual user name or password that is retrieved from the current host security domain is shorter than the replacement sequence, it is padded out with the first character of the **3270SSOPadByte** string used as a padding character. If the actual host user name or password string is longer than the string that is scanned for, these strings are truncated to the length of the scanned string so that the data message length is not affected.

Note that since the user name and password can be sent in any order, the registry string values for the **3270SSOUserTag** and **3270SSOPwdTag** entries must be different for single sign-on to function properly.

The SNA node monitors the SSCP-LU session for these special tag strings at all times and replaces all occurrences of these strings with corresponding looked-up data. On the LU-LU session, the node starts monitoring at start of session (BIND). The node stops monitoring when it has received **3270SSOPostReplaceCount** chains of RUs without seeing a substitution tag. The node will not restart monitoring until it receives an UNBIND–BIND sequence for that session.

Note that the node considers the sequence:

```
BIND, data, UNBIND(BIND FORTHCOMING), BIND …
```

as a continuation of the same LU-LU session and does not restart monitoring on receipt of the second BIND. This sequence is often used by host session managers handing off a session to an application subsystem, and is considered a single terminal session.

User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the node has received **3270SSOPostReplaceCount** chains of RUs without seeing a substitution tag or a timer expires. By default the timer is set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node. After the timer expires, the node will stop scanning messages for the 3270 replacements strings for the user ID and password. If the replacement strings arrive after the timer expires, the replacement strings will be sent to the host unmodified causing the signon to fail. The application will not receive any notification that the timer has expired. The only indication of a problem will likely be that the signon to the host session has failed.

Note that all strings are specified in the registry in ASCII, but the node translates them to EBCDIC through AE character mapping before scanning for a match.

LUA Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide lists the verbs, extensions, control blocks, and return codes that describe the Logical Unit Application programming interface (LUA).

This section contains:

LUA RUI Verbs

This section describes the LUA Request Unit Interface (RUI) verbs. It provides the following information for each RUI verb:

- Details of the LUA verb control block (VCB) structure.
- A description of the verb and its purpose.
- Parameters (VCB structure members) supplied to and returned by LUA. The description of each parameter includes information on the valid values for that parameter.
- Interaction with other verbs.

There are six RUI verbs:

RUI_BID
: Notifies the RUI application that a message is waiting to be read using **RUI_READ**.

RUI_INIT
: Transfers control of the specified LU to the LUA application and establishes a session between the system services control point (SSCP) and the specified LU.

RUI_PURGE
: Cancels a previous **RUI_READ**.

RUI_READ
: Receives responses, SNA commands, and data into an LUA application's buffer.

RUI_TERM
: Ends both the LU session and the SSCP session for a given LUA LU.

RUI_WRITE
: Sends an SNA RU from the LUA application to the host over either the LU session or the SSCP session, and sends responses, SNA commands, and data from an LUA application to the host LU.

The verb descriptions in this section include parameter values specific to each verb. For a complete description of the VCB structure for both RUI and SLI verbs, see LUA Verb Control Blocks.

RUI_BID

The **RUI_BID** verb notifies the RUI application that a message is waiting to be read using RUI_READ.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_BID**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **RUI_BID**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    unsigned char lua_peek_data[12];
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_BID.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_BID only requires this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid

Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length

Not used by **RUI_BID** and should be set to zero.

lua_data_length

Returned parameter. Specifies the length of data returned in **lua_peek_data** for **RUI_BID**.

lua_data_ptr

Pointer to the application-supplied buffer that contains the data to be sent for SLI_SEND and RUI_WRITE or that will receive data for SLI_RECEIVE and RUI_READ. For other RUI and SLI verbs, this parameter is not used and should be set to zero.

lua_post_handle

Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th

Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are set for write functions and returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh

Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for the write function and returned by the read and bid functions. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

lua_message_type

Returned parameter. Specifies the type of SNA message indicated to **RUI_BID**. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

The SLI receives and responds to the BIND, CRV, and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that **RUI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Reserved and should be set to zero.

lua_peek_data
The union member of **LUA_SPECIFIC** used by the **RUI_BID** and SLI_BID verbs. Returned parameter. Contains up to 12 bytes of the data waiting to be read.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_CANCELED
Primary return code; the verb did not complete successfully because it was canceled by another verb.

LUA_TERMINATED

Secondary return code; RUI_TERM was issued while this verb was pending.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BID_ALREADY_ENABLED

Secondary return code; **RUI_BID** was rejected because a previous **RUI_BID** was already outstanding. Only one **RUI_BID** can be outstanding at any one time.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For a Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record, or a parameter not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_RUI_SESSION

Secondary return code; RUI_INIT has not yet completed successfully for the LU name specified on this verb.

LUA_UNSUCCESSFUL
Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_INVALID_PROCESS

Secondary return code; the process that issued this verb was not the same process that issued **RUI_INIT** for this session. Only the process that started a session can issue verbs on that session.

LUA_NEGATIVE_RSP
Primary return code; LUA detected an error in the data received from the host. Instead of passing the received message to the application on RUI_READ, LUA discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host.

LUA informs the application on a subsequent **RUI_READ** or **RUI_BID** that a negative response was sent.

The secondary return code contains the sense code sent to the host on the negative response. See SNA Considerations Using LUA for information on interpreting the sense code values that can be returned.

A zero secondary return code indicates that, following a previous RUI_WRITE of a negative response to a message in the middle of a chain, LUA has now received and discarded all messages from this chain.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session has failed because of a problem with the link service or with the host LU.

LUA_RUI_LOGIC_ERROR

Secondary return code; an internal error was detected within LUA. This error should not occur during normal operation.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

RUI_BID is used by applications that require notification that a message is waiting to be read. This allows the application to determine how it will handle the message before issuing RUI_READ.

When a message is available, **RUI_BID** returns with details of the message flow on which it was received, the message type, the TH and RH of the message, and up to 12 bytes of message data.

The main difference between **RUI_BID** and **RUI_READ** is that **RUI_BID** allows the application to check the data without removing it from the incoming message queue, so it can be left and accessed later. **RUI_READ** removes the message from the queue, so when the application reads the data it must also process it.

Note the following when using **RUI_BID**:

- RUI_INIT must complete successfully before this verb is issued.
- Only one **RUI_BID** can be outstanding at any one time.
- After **RUI_BID** has completed successfully, it can be reissued by setting **lua_flag1.bid_enable** on a subsequent RUI_READ. If the verb is reissued in this way, the application must not free or modify the storage associated with the **RUI_BID** record.
- If a message arrives from the host when **RUI_READ** and **RUI_BID** are both outstanding, **RUI_READ** completes and **RUI_BID** is left in progress.

Each message that arrives is bid only once. After **RUI_BID** indicates that data is waiting on a particular session flow, the application issues **RUI_READ** to receive the data. Any subsequent **RUI_BID** does not report data arriving on that session flow until

the message that was bid has been accepted by issuing **RUI_READ**.

In general, the **lua_data_length** parameter returned on this verb indicates only the length of data in **lua_peek_data**, not the total length of data on the waiting message (except when a value of less than 12 is returned). The application should ensure that the data length on **RUI_READ** that accepts the data is sufficient to contain the message.

See also

RUI_INIT, RUI_READ, RUI_TERM, RUI_WRITE, SLI_OPEN

RUI_INIT

The **RUI_INIT** verb transfers control of the specified LU to the Windows LUA application. **RUI_INIT** establishes a session between the SSCP and the specified LU.

> **Note** For 3270 emulator users, a Microsoft® Host Integration Server or Microsoft® SNA Server extension has been added that allows you to use 3270 LUs rather than the LUA LUs. For more information, see Remarks in this topic.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_INIT**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_INIT.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_INIT requires this parameter.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Returned parameter. Specifies the session identifier.

lua_max_length
Not used by **RUI_INIT** and should be set to zero.

lua_data_length
Not used by **RUI_INIT** and should be set to zero.
lua_data_ptr
Not used by **RUI_INIT** and should be set to zero.
lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **RUI_INIT** and should be set to zero.
lua_rh
Not used by **RUI_INIT** and should be set to zero.
lua_flag1
Not used by **RUI_INIT** and should be set to zero.
lua_message_type
Specifies the type of the inbound or outbound SNA commands and data. This is a returned parameter for **RUI_INIT**. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

The SLI receives and responds to the BIND, CRV, and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1. (Note that **RUI_INIT** always completes asynchronously unless it returns an error such as LUA_PARAMETER_CHECK.)

lua_resv56
Supplied parameter. A reserved field used by **RUI_INIT** and SLI_OPEN. All other reserved fields in the array must be left blank. See the discussion of these Host Integration Server or SNA Server extensions in the Remarks section.

lua_resv56[1]

Supplied parameter. Indicates whether an RUI application can access LUs configured as 3270 LUs, in addition to LUA LUs. If this parameter is nonzero, 3270 LUs can be accessed.

lua_resv56[2]

Supplied parameter. Indicates whether the RUI library will release the LU when the LU-SSCP session or connection goes away. If this parameter is nonzero, the LU will not be released.

lua_resv56[3]

Supplied parameter. Indicates whether incomplete reads are supported. If this parameter is set to a nonzero value, incomplete or truncated reads are supported. See the remarks for RUI_READ for more details.

lua_resv56[4]

Supplied parameter. Indicates whether the RUI library will allow the application to keep hold of the LU if it is recycled at the host. If this parameter is nonzero, the application can keep hold of the LU.

lua_encr_decr_option
Field for cryptography options. On **RUI_INIT**, only the following are supported:

- **lua_encr_decr_option** = 0
- **lua_encr_decr_option** = 128

Values from 1 through 127 are not supported.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.
LUA_CANCELED
Primary return code; the verb did not complete successfully because it was canceled by another verb.

LUA_TERMINATED

Secondary return code; RUI_TERM was issued before **RUI_INIT** completed.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; the **lua_luname** parameter did not match any LUA LU name or LU pool name in the configuration file.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For a Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record, or a parameter not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_DUPLICATE_RUI_INIT

Secondary return code; the **lua_luname** parameter specified an LU name or LU pool name already in use by this application (or for which this application already has **RUI_INIT** in progress).

LUA_UNSUCCESSFUL
Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_COMMAND_COUNT_ERROR

Secondary return code, which indicates one of the following errors occurred:

The verb could not be issued because the application had already reached its maximum number of active sessions. On Windows 2000, Windows NT, Windows 98, and Windows 95, an application can have as many as 15,000 sessions active at any time. In the Windows 3.x environment, an application can have as many as 16 sessions active at any time. In OS/2, an application can have as many as 512 sessions active at any time.

The verb specified the name of an LU pool or the name of an LU in a pool, but all the LUs in the pool are in use.

LUA_ENCR_DECR_LOAD_ERROR

Secondary return code; the verb specified a value for **lua_encr_decr_option** other than 0 or 128.

LUA_INVALID_PROCESS

Secondary return code; the LU specified by **lua_luname** is in use by another process.

LUA_LINK_NOT_STARTED

Secondary return code; the connection to the host has not been started; none of the link services it could use are active.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session failed because of a problem with the link service or with the host LU.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

This verb must be the first LUA verb issued for the session. Until this verb has completed successfully, the only other LUA verb that can be issued for this session is RUI_TERM (which terminates a pending **RUI_INIT**).

All other verbs issued on this session must identify the session using one of the following parameters from this verb:

- The session identifier, returned to the application in **lua_sid**.
- The LU name or LU pool name, supplied by the application in the **lua_luname** parameter.

RUI_INIT completes after an ACTLU message is received from the host. If necessary, the verb waits indefinitely. If an ACTLU has already been received prior to **RUI_INIT**, LUA sends a NOTIFY to the host to inform it that the LU is ready for use. Note that neither ACTLU nor NOTIFY is visible to the LUA application.

After **RUI_INIT** has completed successfully, this session uses the LU for which the session was started. No other LUA session (from this or any other application) can use the LU until **RUI_TERM** is issued, or until an LUA_SESSION_FAILURE primary return code is received.

Using 3270 LUs

To provide 3270 emulator users the ability to use the Emulator Interface Specification (EIS) configuration call with the RUI API, a Host Integration Server or SNA Server extension has been added to the RUI. This extension allows you to use 3270 LUs rather than LUA LUs. If an application sets **lua_resv56[1]** to a nonzero value on the **RUI_INIT** call then 3270 LUs can be used.

Don't Release the LU

If an application sets **lua_resv56[2]** to a nonzero value on the **RUI_INIT** call then the RUI library will not release the LU when the LU-SSCP session or connection goes away. When this Host Integration Server or SNA Server extension is enabled, the application does not have to issue a new **RUI_INIT** after a session failure or connection failure. When the LU-SSCP session comes back up (the application can use **WinRUIGetLastInitStatus** to detect this), the application can start using it again.

Support Chunking on this Session

If an application sets **lua_resv56[3]** to a nonzero value on the **RUI_INIT** session establishment, this enables a Host Integration Server or SNA Server extension that can change the behavior of **RUI_READ**. The default behavior for an **RUI_READ** call is to truncate data (discarding any data remaining) if the application's data buffer is not large enough for receive all of the data in the RU, returning an error code. When **lua_resv56[3]** is set to a nonzero value on the **RUI_INIT** call, then an **RUI_READ** issued where the application's data buffer is not large enough will not result in the RU data being discarded. The **RUI_READ** verb will return success (LUA_OK) for the primary return code and LUA_DATA_INCOMPLETE for the secondary return code. Subsequent **RUI_READ** requests can then be issued to retrieve the data that exceeded the application's data buffer.

Ignore DACTLUs

If an application sets **lua_resv56[4]** to a nonzero value on the **RUI_INIT** session establishment, this enables a Host Integration Server or SNA Server extension and the RUI library will allow the application to keep hold of the LU if it is recycled at the host (that is, deactivated and reactivated).

> **Note** All other reserved fields must be left blank.

For more information, see the description of the sepdcrec function in the section of the SDK documentation on the 3270 Emulator Interface Specification..

Encryption

Session-level cryptography is implemented through Cryptography Verification (CRV) requests; RUI applications must perform all necessary processing of these requests. For all interfaces other than RUI, CRV requests are rejected with a negative response by the Host Integration Server or SNA Server.

For **RUI_INIT**, the following options are supported:

- **lua_encr_decr_option** = 0
- **lua_encr_decr_option** = 128

Values from 1 through 127 (ACSRENCR and ACSROECR routines) are not supported.

The sending application is responsible for padding data to a multiple of eight bytes and for setting the padded data indicator bit in the RH as well as for encryption. The receiving application is responsible for removing the padding after decryption.

See also

RUI_INIT, RUI_TERM, SLI_OPEN

RUI_PURGE

The **RUI_PURGE** verb cancels a previous RUI_READ.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_PURGE**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *      lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_PURGE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_PURGE only requires this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **RUI_PURGE** and should be set to zero.

lua_data_length

Not used by **RUI_PURGE** and should be set to zero.

lua_data_ptr
Points to the location of the RUI_READ verb's VCB that is to be canceled.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **RUI_PURGE** and should be set to zero.

lua_rh
Not used by **RUI_PURGE** and should be set to zero.

lua_flag1
Not used by **RUI_PURGE** and should be set to zero.

lua_message_type
Not used by **RUI_PURGE** and should be set to zero.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Reserved and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_CANCELED
Primary return code; the verb did not complete successfully because it was canceled by another verb.

LUA_TERMINATED

Secondary return code; RUI_TERM was issued while **RUI_PURGE** was pending.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter was set to null.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For a Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record, or a parameter not used by this verb, was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_RUI_SESSION

Secondary return code; RUI_INIT has not yet completed successfully for the LU name specified on this verb.

LUA_UNSUCCESSFUL
Primary return code; the verb supplied was valid, but the verb did not complete successfully.

LUA_INVALID_PROCESS

Secondary return code; the OS/2 process that issued this verb was not the same process that issued **RUI_INIT** for this session. Only the process that started a session can issue verbs on that session.

LUA_NO_READ_TO_PURGE

Secondary return code; either **lua_data_ptr** did not contain a pointer to an RUI_READ VCB, or **RUI_READ** completed before **RUI_PURGE** was issued.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node was broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session failed because of a problem with the link service or with the host LU.

LUA_RUI_LOGIC_ERROR

Secondary return code; an internal error was detected within LUA. This error should not occur during normal operation.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

RUI_READ can wait indefinitely if it is sent without using the **lua_flag1.nowait** (immediate return) option and no data is available on the specified flow; **RUI_PURGE** forces the waiting verb to return (with the primary return code LUA_CANCELED).

This verb is used only when **RUI_READ** has been issued and is pending completion (that is, the primary return code is LUA_IN_PROGRESS).

See also

RUI_INIT, RUI_READ, RUI_TERM, RUI_WRITE, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

RUI_READ

The **RUI_READ** verb receives responses, SNA commands, and data into a Windows LUA application's buffer.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_READ**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_READ.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_READ only requires this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Specifies the length of received buffer for **RUI_READ** and SLI_RECEIVE. Not used by other RUI and SLI verbs and should be set to zero.

lua_data_length
Returned parameter. Specifies the length of data returned in **lua_peek_data** for the RUI_BID verb.

lua_data_ptr
Pointer to the application-supplied buffer that is to receive the data from an **RUI_READ** verb. Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

When **RUI_READ** is issued, this parameter points to the location to receive the data from the host.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message sent or received. Various subparameters are set for write functions and returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for the write function and returned by the read and bid functions. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set **lua_flag1.nowait** to 1 to indicate that you want **RUI_READ** to return immediately whether or not data is available to be read, or set it to zero if you want the verb to wait for data before returning.

Set **lua_flag1.bid_enable** to 1 to re-enable the most recent RUI_BID (equivalent to issuing **RUI_BID** again with exactly the same parameters as before), or set it to zero if you do not want to re-enable **RUI_BID**. Note that re-enabling the previous **RUI_BID** reuses the VCB originally allocated for it, so this VCB must not have been freed or modified.

Set one or more of the following flags to 1 to indicate from which message flow to read data:

lua_flag1.sscp_exp

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

If more than one flag is set, the highest-priority data available is returned. The order of priorities (highest first) is: SSCP expedited, LU expedited, SSCP normal, LU normal. The equivalent flag in the **lua_flag2** group is set to indicate from which flow the data was read.

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. Returned parameter. Specifies the type of SNA message indicated to **RUI_READ**. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that RUI_BID was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.
lua_encr_decr_option
Reserved and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_DATA_INCOMPLETE

Secondary return code; **RUI_READ** was not able to return all of the data received because the application's data buffer (indicated by **lua_max_length**) was not large enough. Subsequent **RUI_READ** requests can be issued to retrieve the remaining RUI data.

Note that this is not the default behavior for **RUI_READ** and is only enabled when **lua_resv56[3]** is set to a nonzero value in the verb control block when calling RUI_INIT during session establishment. See Remarks for more details.

LUA_CANCELED
Primary return code; the verb did not complete successfully because it was canceled by another verb or by an internal error.

LUA_PURGED

Secondary return code; **RUI_READ** has been canceled by RUI_PURGE.

LUA_TERMINATED

Secondary return code; RUI_TERM was issued while **RUI_READ** was pending.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter contained an invalid value.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BID_ALREADY_ENABLED

Secondary return code; **lua_flag1.bid_enable** was set to re-enable RUI_BID but the previous **RUI_BID** was still in progress.

LUA_DUPLICATE_READ_FLOW

Secondary return code; the flow flags in the **lua_flag1** group specified one or more session flows for which **RUI_READ** was already outstanding. Only one **RUI_READ** at a time can be waiting on each session flow.

LUA_INVALID_FLOW

Secondary return code; none of the **lua_flag1** flow flags was set. At least one of these flags must be set to 1, to indicate from which flow or flows to read.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For a Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_NO_PREVIOUS_BID_ENABLED

Secondary return code; **lua_flag1.bid_enable** was set to re-enable RUI_BID, but there was no previous **RUI_BID** that could be enabled. (See Remarks for more information.)

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record or a parameter not used by this verb was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_RUI_SESSION

Secondary return code; RUI_INIT has not yet completed successfully for the LU name specified on **RUI_READ**.

LUA_NEGATIVE_RSP
Primary return code; indicates one of the following two cases, which can be distinguished by the secondary return code:

- LUA detected an error in the data received from the host. Instead of passing the received message to the application on **RUI_READ**, LUA discards the message (and the rest of the chain if it is in a chain), and sends a negative response to the host. LUA informs the application on a subsequent **RUI_READ** or RUI_BID that a negative response was sent.
- The LUA application previously sent a negative response to a message in the middle of a chain. LUA has purged subsequent messages in this chain, and is now reporting to the application that all messages from the chain have been received and purged.

LUA_SEC_RC

Secondary return code; this parameter is a nonzero secondary return code containing the sense code sent to the host on the negative response. This indicates that LUA detected an error in the host data and sent a negative response to the host. See SNA Considerations Using LUA for information on interpreting the sense code values that may be returned.

A secondary return code of zero indicates that, following a previous RUI_WRITE of a negative response to a message in the middle of a chain, LUA has now received and discarded all messages from this chain.

LUA_UNSUCCESSFUL
Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_DATA_TRUNCATED

Secondary return code; the **lua_data_length** parameter was smaller than the actual length of data received on the message. Only **lua_data_length** bytes of data were returned to the verb; the remaining data was discarded. Additional parameters are also returned if this secondary return code is obtained.

LUA_NO_DATA

Secondary return code; **lua_flag1.nowait** was set to indicate immediate return without waiting for data, and no data was currently available on the specified session flow or flows.

LUA_INVALID_PROCESS

Secondary return code; the OS/2 process that issued this verb was not the same process that issued RUI_INIT for this session. Only the process that started a session can issue verbs on that session.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node was broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session failed because of a problem with the link service or with the host LU.

LUA_RUI_LOGIC_ERROR

Secondary return code; an internal error was detected within LUA. This error should not occur during normal operation.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

RUI_INIT must have completed successfully before **RUI_READ** is issued.

While an existing **RUI_READ** is pending, you can issue another **RUI_READ** only if it specifies a different session flow or flows from pending **RUI_READ** verbs; that is, you cannot have more than one **RUI_READ** outstanding for the same session flow.

You can specify a particular message flow (LU normal, LU expedited, SSCP normal, or SSCP expedited) from which to read data, or you can specify more than one message flow. You can have multiple **RUI_READ** verbs outstanding, provided that no two of them specify the same flow.

Data is received by the application on one of four session flows. The four session flows, from highest to lowest priority are:

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

The data flow type that **RUI_READ** is to process is specified in the **lua_flag1** parameter. The application can also specify whether it wants to look at more than one type of data flow. When multiple flow bits are set, the highest priority is received first. When **RUI_READ** completes processing, **lua_flag2** indicates the specific type of flow for which data has been received by the Windows LUA application.

If RUI_BID successfully completes before an **RUI_READ** is issued, the Windows LUA interface can be instructed to reuse the last **RUI_BID** verb's VCB. To do this, issue the **RUI_READ** with **lua_flag1.bid_enable** set.

The **lua_flag1.bid_enable** parameter can be used only if the following are true:

- **RUI_BID** has already been issued successfully and has completed.
- The storage allocated for **RUI_BID** has not been freed or modified.
- No other **RUI_BID** is pending.

When using **lua_flag1.bid_enable**, the **RUI_BID** storage must not be freed because the last **RUI_BID** verb's VCB is used. Also, when using **lua_flag1.bid_enable**, the successful completion of **RUI_BID** will be posted.

If **RUI_READ** is issued with **lua_flag1.nowait** when no data is available to receive, LUA_NO_DATA will be the secondary return code set by the Windows LUA interface.

If the data received is longer than **lua_max_length**, it is truncated; and only **lua_max_length** bytes of data are returned. The primary return code LUA_UNSUCCESSFUL and the secondary return code LUA_DATA_TRUNCATED are also returned. The RUI library returns as much data as possible to the application's data buffer, but the remaining data in the RUI is discarded and cannot be extracted on subsequent **RUI_READ** requests. This forces the RUI application to allocate an **RUI_READ** data buffer large enough to handle the full RU size.

This default behavior can be changed by setting the value of **lua_resv56[3]** to a nonzero value in the verb control block when calling RUI_INIT during session establishment. In this case, if the data received is longer than **lua_max_length**, an **RUI_READ** request will return a primary return code of LUA_OK and a secondary return code of LUA_DATA_INCOMPLETE. An RUI application can then issue new **RUI_READ** calls and receive the remainder of the data. Note that this enhancement has not been adopted as part of the Microsoft Windows Open Services Architecture (WOSA) LUA API standard and differs from the implementation of RUI by IBM.

After a message has been read using **RUI_READ,** it is removed from the incoming message queue and cannot be accessed again. (Note that RUI_BID can be used as a nondestructive read; that is, the application can use it to check the type of data available, but the data remains on the incoming queue and does not need to be used immediately.)

Pacing can be used on the primary-to-secondary half-session (this is specified in the host configuration), to protect the LUA application from being flooded with messages. If the LUA application is slow to read messages, Host Integration Server or SNA Server delays the sending of pacing responses to the host to slow it down.

See also

RUI_BID, RUI_INIT, RUI_TERM, RUI_WRITE, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

RUI_TERM

The **RUI_TERM** verb ends both the LU session and the SSCP session for a given LUA LU.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_TERM**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_TERM.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_TERM only requires this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be set to zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **RUI_TERM** and should be set to zero.

lua_data_length

Not used by **RUI_TERM** and should be set to zero.

lua_data_ptr
Not used by **RUI_TERM** and should be set to zero.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **RUI_TERM** and should be set to zero.

lua_rh
Not used by **RUI_TERM** and should be set to zero.

lua_flag1
Not used by **RUI_TERM** and should be set to zero.

lua_message_type
Not used by **RUI_TERM** and should be set to zero.

lua_flag2
Not used by **RUI_TERM** and should be set to zero.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Reserved and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For a Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record or a parameter not used by this verb was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_RUI_SESSION

Secondary return code; RUI_INIT has not yet completed successfully for the LU name specified on **RUI_TERM**.

LUA_UNSUCCESSFUL
Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_COMMAND_COUNT_ERROR

Secondary return code; **RUI_TERM** was already pending when the verb was issued.

LUA_INVALID_PROCESS

Secondary return code; the OS/2 process that issued this verb was not the same process that issued **RUI_INIT** for this session. Only the process that started a session can issue verbs on that session.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node was broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session failed because of a problem with the link service or with the host LU.

LUA_RUI_LOGIC_ERROR

Secondary return code; an internal error was detected within LUA. This error should not occur during normal operation.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

This verb can be issued at any time after RUI_INIT has been issued (whether or not it has completed). If any other LUA verb is pending when **RUI_TERM** is issued, no further processing on the pending verb takes place, and it returns with a primary return code of LUA_CANCELED.

After this verb has completed, no other LUA verb can be issued for this session.

See also

RUI_INIT, SLI_OPEN

RUI_WRITE

The **RUI_WRITE** verb sends an SNA request or response unit from the LUA application to the host over either the LU session or the SSCP session, and sends responses, SNA commands, and data from a Windows LUA application to the host LU.

The following structure describes the **LUA_COMMON** member of the VCB used by **RUI_WRITE**.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_RUI_WRITE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

RUI_WRITE only requires this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero.

lua_cobol_offset
Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **RUI_WRITE** and should be set to zero.

lua_data_length
Returned parameter. Specifies the length of data returned in **lua_peek_data** for the RUI_BID verb.

lua_data_ptr
Points to the buffer containing the data to be sent to the host by **RUI_WRITE**.

Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message sent or received. Various subparameters are set for write functions and returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. For the RH for **RUI_WRITE**, all fields except the queued-response indicator (**lua_rh.qri**) and pacing indicator (**lua_rh.pi**) are used. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set one of the following flags to 1 to indicate on which message flow the data is to be sent:

lua_flag1.sscp_exp

lua_flag1.sscp_norm

lua_flag1.lu_exp

lua_flag1.lu_norm

lua_message_type
Not used by **RUI_WRITE** and should be set to zero.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that RUI_BID was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Reserved and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_CANCELED
Primary return code; the verb did not complete successfully because it was canceled by another verb.

LUA_TERMINATED

Secondary return code; the verb was canceled because RUI_TERM was issued for this session.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter contained an invalid value.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_DUPLICATE_WRITE_FLOW

Secondary return code; **RUI_WRITE** was already outstanding for the session flow specified on this verb (the session flow is specified by setting one of the **lua_flag1** flow flags to 1). Only one **RUI_WRITE** at a time can be outstanding on each session flow.

LUA_INVALID_FLOW

Secondary return code; the **lua_flag1.sscp_exp** flow flag was set, indicating that the message should be sent on the SSCP expedited flow. LUA does not allow applications to send data on this flow.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Windows 2000, Windows NT, Windows 98, or Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the Windows LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the Windows LUA VCB does not contain a valid semaphore or queue handle, which is needed when the verb completes asynchronously.

LUA_MULTIPLE_WRITE_FLOWS

Secondary return code; more than one of the **lua_flag1** flow flags was set to 1. One and only one of these flags must be set to 1, to indicate which session flow the data is to be sent on.

LUA_REQUIRED_FIELD_MISSING

Secondary return code; indicates one of the following cases:

- None of the **lua_flag1** flow flags was set. One and only one of these flags must be set to 1.
- **RUI_WRITE** was used to send a response, and the response required more data than was supplied.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved field in the verb record or a parameter not used by this verb was set to a nonzero value.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_MODE_INCONSISTENCY

Secondary return code; the SNA message sent on **RUI_WRITE** was not valid at this time. This is caused by trying to send data on the LU session before the session is bound. Check the sequence of SNA messages sent.

LUA_NO_RUI_SESSION

Secondary return code; RUI_INIT has not yet completed successfully for the LU name specified on this verb.

LUA_UNSUCCESSFUL

Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; indicates one of the following cases:

- The **lua_rh.fi** bit (format indicator) was set to 1, but the first byte of the supplied RU was not a recognized request code.
- The **lua_rh.ruc** parameter (RU category) specified the network control (NC) category; LUA does not allow applications to send requests in this category.

LUA_INVALID_PROCESS

Secondary return code; the OS/2 process that issued this verb was not the same process that issued **RUI_INIT** for this session. Only the process that started a session can issue verbs on that session.

LUA_INVALID_SESSION_PARAMETERS

Secondary return code; the application used **RUI_WRITE** to send a positive response to a BIND message received from the host. However, Host Integration Server or SNA Server cannot accept the BIND parameters as specified, and has sent a negative response to the host. See SNA Considerations Using LUA for more information on the BIND profiles accepted by Host

Integration Server or SNA Server.

LUA_RSP_CORRELATION_ERROR

Secondary return code; when using **RUI_WRITE** to send a response, **lua_th.snf** (which indicates the sequence number of the received message being responded to) did not contain a valid value.

LUA_RU_LENGTH_ERROR

Secondary return code; the **lua_data_length** parameter contained an invalid value. When sending data on the LU normal flow, the maximum length is as specified in the BIND received from the host; for all other flows the maximum length is 256 bytes.

> **Note** Any other secondary return code is an SNA sense code indicating that the supplied SNA data was invalid or could not be sent. See SNA Considerations Using LUA for information on interpreting the SNA sense codes that can be returned.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node was broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_SESSION_FAILURE
Primary return code; a required Host Integration Server or SNA Server component has terminated.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; indicates that the LUA session has failed because of a problem with the link service or with the host LU.

LUA_RUI_LOGIC_ERROR

Secondary return code; an internal error was detected within LUA. This error should not occur during normal operation.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.
LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or has terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

Remarks

RUI_INIT must be issued successfully before this verb is issued.

When sending an SNA request, all applicable values in the **lua_rh** must be set. Chaining and bracketing are the responsibility of the application.

When sending a response, the type of response determines the **RUI_WRITE** information required. For all responses, you must:

- Set the selected **lua_rh.rri** flag to 1.
- Provide the sequence number in **lua_th.snf** for the request to which you are responding.

For multichain message responses, the sequence number of the last received chain element must be used. For a response to a multichain message ending with a CANCEL command, the CANCEL command sequence number is used.

For positive responses that only require the request code, set **lua_rh.ri** to zero (indicating that the response is positive) and **lua_data_length** to zero (indicating that no data is provided). The request code is filled in by the RUI, using the sequence number provided.

For negative responses, set **lua_rh.ri** to 1, **lua_data_ptr** to the SNA sense code address, and **lua_data_length** to the SNA sense code length (four bytes). The sequence number is used by the RUI to fill in the request code.

For positive responses to the BIND and STSN commands that require data in the responses, set **lua_data_ptr** to point to the response and set **lua_data_length** to the length of the data provided in **lua_data_ptr**.

While an existing **RUI_WRITE** is pending, you can issue a second **RUI_WRITE** only if it specifies a different session flow from the pending **RUI_WRITE**. That is, you cannot have more than one **RUI_WRITE** outstanding for the same session flow.

RUI_WRITE can be issued on the SSCP normal flow at any time after a successful RUI_INIT. **RUI_WRITE** verbs on the LU expedited or LU normal flows are permitted only after a BIND has been received, and must abide by the protocols specified on the BIND.

Note that successful completion of **RUI_WRITE** indicates that the message was queued successfully to the data link; it does not necessarily indicate that the message was sent successfully, or that the host accepted it.

Pacing can be used on the secondary-to-primary half-session (specified on the BIND) to prevent the LUA application from sending more data than the local or remote LU can handle. If this is the case, an **RUI_WRITE** on the LU normal flow may be delayed by LUA and may take some time to complete.

See also

RUI_INIT, RUI_READ, RUI_TERM, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

LUA SLI Verbs

This section describes the LUA Session Level Interface (SLI) verbs. It provides the following information for each SLI verb:

- Details of the LUA verb control block (VCB) structure.
- A description of the verb and its purpose.
- Parameters (VCB structure members) supplied to and returned by LUA. The description of each parameter includes information on the valid values for that parameter.
- Interaction with other verbs.

There are six SLI verbs and two user-defined routines:

SLI_BID
: Notifies the SLI application that a message is waiting to be read using **SLI_RECEIVE** and provides the current status of the session to the Windows LUA application.

SLI_CLOSE
: Ends a session opened with **SLI_OPEN**.

SLI_OPEN
: Transfers control of the specified LU to the Windows LUA application and establishes a session between the system services control point (SSCP) and the specified LU, as well as an LU-LU session.

SLI_PURGE
: Cancels **SLI_RECEIVE** verbs issued with a wait condition.

SLI_RECEIVE
: Receives responses, SNA commands, and data into the buffer of a Windows LUA application. **SLI_RECEIVE** also provides the current status of the session to the Windows LUA application.

SLI_RECEIVE_EX
: Receives responses, SNA commands, and data into the buffer of a Windows LUA application and supports inbound chaining allowing sending up to 4,294,967,295 bytes in a single verb. **SLI_RECEIVE_EX** also provides the current status of the session to the Windows LUA application.

SLI_SEND
: Sends responses, SNA commands, and data from a Windows LUA application to a host LU.

SLI_SEND_EX
: Sends responses, SNA commands, and data from a Windows LUA application to a host LU and supports inbound chaining allowing receiving up to 4,294,967,295 bytes in a single verb.

SLI_BIND_ROUTINE
: Notifies the Windows LUA application that a BIND request has come from the host and allows the user-supplied routine to examine the request and formulate a response.

SLI_STSN_ROUTINE
: Notifies the Windows LUA application that an STSN command has come from the host and allows the user-supplied routine to examine the request and formulate a response.

Cryptography is not defined as part of the Windows LUA standard.

The verb descriptions in this section include parameter values specific to each verb. For a complete description of the VCB structure for both RUI and SLI verbs, see LUA Verb Control Blocks.

SLI_BID

The **SLI_BID** verb notifies the SLI application that a message is waiting to be read using SLI_RECEIVE. **SLI_BID** also provides the current status of the session to the Windows LUA application.

The following structure describes the LUA_COMMON member of the VCB used by SLI_BID.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **SLI_BID**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    unsigned char  lua_peek_data[12];
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_BID.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_BID only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_BID** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_BID** and should be set to zero.

lua_data_length
Returned parameter. Specifies the length of data returned in **lua_peek_data**.

lua_data_ptr
Pointer to the application-supplied buffer that contains the data to be sent for SLI_SEND and RUI_WRITE or that will receive data for SLI_RECEIVE and RUI_READ. Not used by other RUI and SLI verbs and should be set to zero.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00** FM data segment
LUA_RH_NC (**0x20** Network control
LUA_RH_DFC (**0x40** Data flow control
LUA_RH_SC (**0x60** Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

lua_message_type

Returned parameter. Specifies the type of SNA message indicated to SLI_BID. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_STSN

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that **SLI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.
lua_encr_decr_option
Not used by **SLI_BID** and should be set to zero.
lua_peek_data
The union member of **LUA_SPECIFIC** used by the RUI_BID and **SLI_BID** verbs. Returned parameter. Contains up to 12 bytes of the data waiting to be read.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** name was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with the value of **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_SLI_BID_PENDING

Secondary return code; an SLI verb was still active when another **SLI_BID** was issued. Only one **SLI_BID** can be active at a time.

LUA_SESSION_FAILURE
Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued, or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_RECEIVER_IN_TRANSMIT_MODE

Secondary return code; either resources needed to handle normal flow data were not available or the state of the half-duplex contention was not received when a normal-flow request was received. The result is a race condition. This SNA sense code is also an exception request sense code.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_CHAINING_ERROR

Secondary return code; the sequence of the chain indicator settings is in error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_BRACKET

Secondary return code; the sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DIRECTION

Secondary return code; while the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_QUIESCED

Secondary return code; a DFC or FMD request was received from a half-session that sent either a SHUTC command or QC command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_NO_BEGIN_BRACKET

Secondary return code; the receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_IMMEDIATE_REQUEST_MODE_ERROR

Secondary return code; the request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_QUEUED_RESPONSE_ERROR

Secondary return code; the request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_ERP_SYNC_EVENT_ERROR

Secondary return code; a violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was

prevented.

LUA_RSP_CORRELATION_ERROR

Secondary return code; a response was sent that does not correspond to a previously received request or a response was received that does not correspond to a previously sent request.

LUA_RSP_PROTOCOL_ERROR

Secondary return code; a violation of the response protocol was found in the response received from the primary half-session.

LUA_BB_NOT_ALLOWED

Secondary return code; the begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EB_NOT ALLOWED

Secondary return code; the end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EXCEPTION_RSP_NOT_ALLOWED

Secondary return code; when an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_DEFINITE_RSP_NOT_ALLOWED

Secondary return code; when a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_ALLOWED

Secondary return code; the change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_RESPONSE_NOT_ALLOWED

Secondary return code; a request other than an EXR contained a "no response." The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CHAINING_NOT_SUPPORTED

Secondary return code; the chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_BRACKETS_NOT_SUPPORTED

Secondary return code; the bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_SUPPORTED

Secondary return code; the change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the

half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_FI

Secondary return code; the format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

Secondary return code; the code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_RU_CATEGORY

Secondary return code; the request unit category indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_REQUEST_CODE

Secondary return code; the request code was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_SPEC_OF_SDI_RTI

Secondary return code; the sense-data-included indicator (SDI) and the response-type-indicator (RTI) were not specified correctly on a response. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_DR1I_DR2I_ERI

Secondary return code; the definite response 1 indicator (DR1I), the definite response 2 indicator (DR2I), and the exception response indicator (ERI) were specified incorrectly. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_QRI

Secondary return code; the queued response indicator (QRI) was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_EDI

Secondary return code; the enciphered data indicator (EDI) was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_PDI

Secondary return code; the padded data indicator (PDI) was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_UNSUCCESSFUL

Primary return code; the verb record supplied was valid, but the verb did not complete successfully.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server SNA Server when an LUA verb was issued.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_RECEIVE_CORRELATION_TABLE_FULL

Secondary return code; the session receive correlation table for the flow requested reached its capacity.

LUA_NEGATIVE_RESPONSE

Primary return code; either LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; the LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_DATA_TRAFFIC_RESET

Secondary return code; a half-session of an active session but with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_NOT_RESET

Secondary return code; while the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_SC_PROTOCOL_VIOLATION

Secondary return code; a violation of the session control (SC) protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_INVALID_SC_OR_NC_RH

Secondary return code; the RH of an SC or NC request was invalid.

LUA_PACING_NOT_SUPPORTED

Secondary return code; the request contained a pacing indicator when support of pacing for this session does not exist for the receiving half-session or boundary function half-session. The BIND options chosen previously or the architectural rules were violated by **lua_rh** values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NAU_INOPERATIVE

Secondary return code; the network addressable unit (NAU) is not able to process responses or requests. Delivery to the receiver could not take place for one of the following reasons:

A path information unit error

A path outage

An invalid sequence of requests for activation

If a path error is received during an active session, that usually means there is no longer a valid path to the session partner.

LUA_CANCELED

Primary return code; the secondary return code gives the reason for canceling the command.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process has been canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.

LUA_STATUS
Primary return code; the secondary return code contains SLI status information for the application.

LUA_READY

Secondary return code; following a NOT_READY status, this status is issued to notify you that the SLI is ready to process commands.

LUA_NOT_READY

Secondary return code; an SNA UNBIND type 0x02 command was received, which means a new BIND is coming.

- If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command.
- If the UNBIND type 0x02 is received while SLI_OPEN is still processing, the primary return code is session-failure, not status. Or, the receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_INIT_COMPLETE

Secondary return code; the LUA interface initialized the session while **SLI_OPEN** was processing. LUA applications that issue **SLI_OPEN** with the **lua_open_type_prim_sscp** parameter receive this status on SLI_RECEIVE or **SLI_BID**.

LUA_SESSION_END_REQUESTED

Secondary return code; the LUA interface received an SNA shutdown command (SHUTD) from the host, which means the host is ready to shut down the session.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_BID does the following:

- Notifies a Windows LUA application that a message is waiting to be read.
- Provides the current session status.
- Provides a preview of the next message that will be read by SLI_RECEIVE.

  This preview contains a maximum of 12 bytes of information (peek data) that enables the Windows LUA application to define its processing strategy for the data.

To use **SLI_BID** within a Windows LUA application, issue **SLI_BID**. When the verb completes, it can be reactivated in the following two ways:

- Reissue **SLI_BID**.
- Issue SLI_RECEIVE with lua_flag1_bid_enable set to 1. This issues an SLI_BID that uses the most recently accepted address for the VCB and establishes the active bid.

Each session can have only one **SLI_BID** at a time.

If multiple messages are available when a Windows LUA application issues SLI_BID, the data flow with the highest priority is returned. The order in which the data can be returned is as follows:

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

If SLI_RECEIVE has flags set to read more than one type of message flow, the data returned by **SLI_BID** might be for a flow different than the one for which you actually receive data through **SLI_RECEIVE**. This situation occurs when higher priority data arrives from the host after **SLI_BID** completes processing, but before **SLI_RECEIVE** is issued.

To ensure that SLI_RECEIVE reads the data, the SLI_BID returned specifies the flow that matches lua_flag2 returned by the completed SLI_BID.

Session Status Return Values

If LUA_STATUS is the primary return code, the secondary return code can be LUA_READY, LUA_NOT_READY, LUA_SESSION_END_REQUESTED, or LUA_INIT_COMPLETE. In addition, if LUA_STATUS is the primary return code, the following parameters are used:

lua_sec_rc

lua_sid

LUA_READY is returned after LUA_NOT_READY status, and indicates that the SLI is again ready to perform all commands.

LUA_NOT_READY indicates that the SLI session is suspended because the SLI has received either an SNA CLEAR command or an SNA UNBIND command with an 0x02 UNBIND type (UNBIND with BIND forthcoming). Depending on what caused the suspension, the session can be reactivated as follows:

- When the suspension is caused by an SNA CLEAR, receiving an SNA SDT reactivates the session.
- When an SNA UNBIND type BIND forthcoming causes suspension of the session and the SLI_OPEN that opened the session is completed, the session is suspended until the SLI receives a BIND and SDT command. The session can also optionally receive an STSN command. As a result, user-supplied routines issued with the initial SLI_OPEN must be re-entered because they will be recalled.

The application can send SSCP data after a CLEAR or UNBIND type BIND forthcoming arrives and before the NOT READY status is read. The application can send and receive SSCP data after reading a NOT READY.

When an SNA UNBIND type BIND forthcoming arrives before completion of the SLI_OPEN that opened the session, LUA_SESSION_FAILURE (not LUA_STATUS) is the primary return code.

LUA_SESSION_END_REQUESTED indicates that the application received an SNA SHUTD from the host. The Windows LUA application should issue SLI_CLOSE to close the session when convenient.

LUA_INIT_COMPLETE is returned only when lua_init_type for SLI_OPEN is LUA_INIT_TYPE_PRIM_SSCP. The status means that SLI_OPEN has been processed sufficiently to allow SSCP data to now be sent or received.

Exception Requests

If a host application request unit is converted into an EXR, sense data will be returned. When an **SLI_BID** completes with the returned verb parameters set as shown, an EXR conversion occurs.

Member	Set to
lua_prim_rc	OK (0x0000)
lua_sec_rc	OK (0x00000000)

lua_rh.rri	bit off (request unit)
lua_rh.sdi	bit on (includes sense data)

Of the seven bytes of data in **lua_peek_data**, bytes 0 through 3 define the error detected. The following table indicates possible sense data and the values of bytes 0 through 3.

Sense data	Value of bytes 0–3
LUA_MODE_INCONSISTENCY	0x08090000
LUA_BRACKET_RACE_ERROR	0x080B0000
LUA_BB_REJECT_NO_RTR	0x08130000
LUA_RECEIVER_IN_TRANSMIT_MODE	0x081B0000
LUA_CRYPTOGRAPHY_FUNCTION_INOP	0x08480000
LUA_SYNC_EVENT_RESPONSE	0x10010000
LUA_RU_DATA_ERROR	0x10020000
LUA_RU_LENGTH_ERROR	0x10020000
LUA_INCORRECT_SEQUENCE_NUMBER	0x20010000

The information returned to bytes 3 through 6 in **lua_peek_data** is determined by the first 3 bytes of the initial request unit that caused the error.

See Also

RUI_INIT, SLI_CLOSE, SLI_OPEN, SLI_RECEIVE

SLI_CLOSE

The **SLI_CLOSE** verb ends a session opened with SLI_OPEN. The LU-LU and LU-SSCP resources are released.

The following structure describes the LUA_COMMON member of the VCB used by SLI_CLOSE.

```
struct LUA_COMMON {
    unsigned short   lua_verb;
    unsigned short   lua_verb_length;
    unsigned short   lua_prim_rc;
    unsigned long    lua_sec_rc;
    unsigned short   lua_opcode;
    unsigned long    lua_correlator;
    unsigned char    lua_luname[8];
    unsigned short   lua_extension_list_offset;
    unsigned short   lua_cobol_offset;
    unsigned long    lua_sid;
    unsigned short   lua_max_length;
    unsigned short   lua_data_length;
    char FAR *       lua_data_ptr;
    unsigned long    lua_post_handle;
    struct LUA_TH    lua_th;
    struct LUA_RH    lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char    lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char    lua_resv56[7];
    unsigned char    lua_encr_decr_option;

};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_CLOSE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_CLOSE only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_CLOSE** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_CLOSE** and should be set to zero.

lua_data_length
Not used by **SLI_CLOSE** and should be set to zero.
lua_data_ptr
Not used by **SLI_CLOSE** and should be set to zero.
lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows 98, or Microsoft® Windows 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **SLI_CLOSE** and should be set to zero.
lua_rh
Not used by **SLI_CLOSE** and should be set to zero.
lua_flag1
Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit. A supplied parameter used by **SLI_CLOSE** to specify whether the session is to be closed immediately (ON) or closed normally (OFF). For verbs other than **SLI_CLOSE**, this flag must be off.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

lua_message_type
Not used by **SLI_CLOSE** and should be set to zero.
lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_resv56
Reserved and should be set to zero.
lua_encr_decr_option
Not used by **SLI_CLOSE** and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows 98, or Microsoft® Windows 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an **SLI_CLOSE** or session failure when a command was issued.

LUA_CLOSE_PENDING

Secondary return code; one of the following has occurred:

- A CLOSE_ABEND was still pending when another CLOSE_ABEND was issued. You can issue a CLOSE_ABEND if a CLOSE_NORMAL is pending.
- Either a CLOSE_ABEND or a CLOSE_NORMAL was still pending when a CLOSE_NORMAL was issued.

LUA_SESSION_FAILURE

Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_UNEXPECTED_SNA_SEQUENCE

Secondary return code; unexpected data or commands were received from the host while SLI_OPEN was processing.

LUA_NEGATIVE_RSP_CHASE

Secondary return code; a negative response to an SNA CHASE command from the host was received by the LUA interface while **SLI_CLOSE** was being processed. **SLI_CLOSE** continued processing to stop the session.

LUA_NEGATIVE_RSP_SHUTC

Secondary return code; a negative response to an SNA SHUTC command from the host was received by the SLI while **SLI_CLOSE** was still being processed. **SLI_CLOSE** continued processing to stop the session.

LUA_NEGATIVE_RSP_SHUTD

Secondary return code; a negative response to an SNA RSHUTD command from the host was received by the LUA interface while **SLI_CLOSE** was still being processed. **SLI_CLOSE** continued processing to stop the session.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a

result, the session was stopped.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued, or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

There are two types of **SLI_CLOSE**: normal and ABEND. For a normal close, **lua_flag1.close_abend** is set to zero. The sequence for a normal close can be initiated either as primary (host-initiated) or secondary (requested by a Windows LUA application). During a primary normal close, the Windows LUA interface:

- Reads the SHUTD command and posts the SESSION_END_REQUESTED status to the application.
- Writes the CHASE command (if necessary).
- Reads and processes the CHASE command response (if necessary).
- Writes the shutdown complete (SHUTC) command.
- Reads and processes the SHUTC command response.
- Reads and processes the CLEAR command (if necessary).
- Writes the CLEAR command response (if necessary).
- Reads and processes the UNBIND command.
- Writes the UNBIND command response.
- Stops the session.

During a secondary normal close, the Windows LUA interface:

- Writes the RSHUTD command.
- Reads and processes the RSHUTD command response.
- Reads and processes the CLEAR command (if necessary).
- Writes the CLEAR command response (if necessary).
- Reads and processes the UNBIND command.
- Writes the UNBIND command response.
- Stops the session.

For an ABEND close, **lua_flag1.close_abend** is set to 1, which directs the Windows LUA interface to close the session immediately. After **SLI_CLOSE** starts processing, the LU-LU connection is terminated and the SSCP is informed that the LU is not capable of sustaining a session.

See Also

SLI_OPEN

SLI_OPEN

The **SLI_OPEN** verb transfers control of the specified LU to the Windows LUA application. **SLI_OPEN** establishes a session between the SSCP and the specified LU, as well as an LU-LU session.

The following structure describes the LUA_COMMON member of the VCB used by SLI_OPEN.

```
struct LUA_COMMON {
    unsigned short   lua_verb;
    unsigned short   lua_verb_length;
    unsigned short   lua_prim_rc;
    unsigned long    lua_sec_rc;
    unsigned short   lua_opcode;
    unsigned long    lua_correlator;
    unsigned char    lua_luname[8];
    unsigned short   lua_extension_list_offset;
    unsigned short   lua_cobol_offset;
    unsigned long    lua_sid;
    unsigned short   lua_max_length;
    unsigned short   lua_data_length;
    char FAR *       lua_data_ptr;
    unsigned long    lua_post_handle;
    struct LUA_TH    lua_th;
    struct LUA_RH    lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char    lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char    lua_resv56[7];
    unsigned char    lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **SLI_OPEN**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    struct union SLI_OPEN open;
};

The SLI_OPEN structure contains the following nested structures and members:
struct LUA_EXT_ENTRY {
    unsigned char lua_routine_type;
    unsigned char lua_module_name[9];
    unsigned char lua_procedure_name[33];
} ;

struct SLI_OPEN {
    unsigned char         lua_init_type;
    unsigned char         lua_resv65;
    unsigned short        lua_wait;
    struct LUA_EXT_ENTRY  lua_open_extension[3];
    unsigned char         lua_ending_delim;
} ;
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode

Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_OPEN.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_OPEN requires this parameter.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Supplied parameter. Specifies the offset from the start of the VCB to the extension list of user-supplied dynamic-link libraries (DLLs). The value must be the beginning of a word boundary unless there is no extension list. In this case, the value must be set to zero.

If this option is not used by SLI_OPEN, then this member should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® SNA Server and should be zero.

lua_sid
Returned parameter. Specifies the session identifier.

lua_max_length
Not used by **SLI_OPEN** and should be set to zero.

lua_data_length
Supplied parameter. Specifies the actual length of the data being sent.

lua_data_ptr
Pointer to the application-supplied buffer that contains the data to be sent for **SLI_OPEN**.

Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

When SLI_OPEN is issued, this parameter can be one of the following:

- The LOGON message for the SSCP normal flow when the initialization type is secondary with an unformatted LOGON message.
- The RU for INITSELF. When the initialization type is secondary with INITSELF, the necessary data for the application is provided.
- For all other open types, this field should be set to zero.

This information is provided by the Windows LUA application.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **SLI_OPEN** and should be set to zero.

lua_rh
Not used by **SLI_OPEN** and should be set to zero.

lua_flag1
Not used by **SLI_OPEN** and should be set to zero.

lua_message_type
Not used by **SLI_OPEN** and should be set to zero.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_resv56
Supplied parameter. Reserved field used by **SLI_OPEN** and RUI_INIT. See Remarks section.

lua_resv56[1]

Supplied parameter. This parameter must be set to 0.

lua_resv56[2]

Supplied parameter. Indicates whether an SLI application can access LUs configured as 3270 LUs, in addition to LUA LUs. If this parameter is set to 1, 3270 LUs can be accessed.

lua_resv56[3]

Supplied parameter. Indicates whether incomplete reads are supported. If this parameter is set to 1, incomplete or truncated reads are supported. See the remarks for RUI_READ for more details.

lua_encr_decr_option

Not used by **SLI_OPEN** and should be set to zero.

open

The union member of **LUA_SPECIFIC** used by **SLI_OPEN**. A supplied set of parameters contained in an **SLI_OPEN** structure required with **SLI_OPEN**.

open.lua_init_type

Supplied parameter. Defines how the LU-LU session is initialized by the Windows LUA interface.

Valid values are as follows:

LUA_INIT_TYPE_SEC_IS

LUA_INIT_TYPE_SEC_LOG

LUA_INIT_TYPE_PRIM

LUA_INIT_TYPE_PRIM_SSCP

open.lua_resv65

Reserved field.

open.lua_wait

Supplied parameter. Represents a secondary retry wait time indicating the number of seconds the Windows LUA interface is to wait before retrying the transmission of the INITSELF or the LOGON message after the host sends any one of these messages:

- A negative response and the secondary return code is one of the following:

  RESOURCE_NOT_AVAILABLE
  (0x08010000)SESSION_LIMIT_EXCEEDED (0x08050000)
  SESSION_SERVICE_PATH_ERROR (0x087D0000)

  Note that SLI_OPEN terminates with error if lua_wait is set to zero and one of the above occurs.

- A network services procedure error (NSPE) message.
- A NOTIFY command, which indicates a procedure error.

open.lua_open_extension

Supplied parameter. Contains a list of application-supplied extension DLLs to process the BIND, STSN, and CRV commands.

open.open_extension.lua_routine_type

The extension routine type. Legal values are:

LUA_ROUTINE_TYPE_BIND

LUA_ROUTINE_TYPE_CRV

LUA_ROUTINE_TYPE_END (indicates end of extension list)

LUA_ROUTINE_TYPE_STSN

open.open_extension.lua_module_name

Supplied parameter. Provides the ASCII module name for the user-supplied extension DLL. The module name can be up to eight characters long, with the remaining bytes set to 0x00.

open.open_extension.lua_procedure_name

Supplied parameter. Provides the procedure name in ASCII for the user-supplied extension DLL. The procedure name can be up to 32 characters long, with the remaining bytes set to 0x00.

open.lua_ending_delim

The extension list delimiter.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** name was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

LUA_DATA_SEGMENT_LENGTH_ERROR

Secondary return code; one of the following occurred:

- The supplied data segment for SLI_RECEIVE or SLI_SEND is not a read/write data segment as required.
- The supplied data segment for SLI_RECEIVE is not as long as that provided in lua_max_length.
- The supplied data segment for SLI_SEND is not as long as that provided in lua_data_length.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_INVALID_OPEN_INIT_TYPE

Secondary return code; the value in the **lua_init_type** contained in **SLI_OPEN** is invalid.

LUA_INVALID_OPEN_DATA

Secondary return code; the **lua_init_type** for the **SLI_OPEN** issued is set to LUA_INIT_TYPE_SEC_IS when the buffer for data does not have a valid INITSELF command.

LUA_INVALID_OPEN_ROUTINE_TYPE

Secondary return code; the **lua_open_routine_type** for the **SLI_OPEN** list of extension routines is invalid.

LUA_DATA_LENGTH_ERROR

Secondary return code; the application did not provide user-supplied data required by the verb issued. Note that when SLI_SEND is issued for an SNA LUSTAT command, status (in four bytes) is required, and that when **SLI_OPEN** is issued with secondary initialization, data is required.

LUA_INVALID_SLI_ENCR_OPTION

Secondary return code; the **lua_encr_decr_option** parameter was set to 128 in **SLI_OPEN**, which is not supported for the encryption/decryption processing option.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_UNEXPECTED_SNA_SEQUENCE

Secondary return code; unexpected data or commands were received from the host while **SLI_OPEN** was processing.

LUA_NEG_RSP_FROM_BIND_ROUTINE

Secondary return code; the user-supplied SLI_BIND routine responded negatively to the BIND. **SLI_OPEN** ended unsuccessfully.

LUA_NEG_RSP_FROM_STSN_ROUTINE

Secondary return code; the user-supplied SLI STSN routine responded negatively to the STSN. **SLI_OPEN** ended unsuccessfully.

LUA_PROCEDURE_ERROR

Secondary return code; a host procedure error is indicated by the receipt of an NSPE or NOTIFY message. The return code is posted to **SLI_OPEN** when the retry option is not used. To use the reset option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue SLI_CLOSE.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued, or some verb other than **SLI_OPEN** was issued before the session was initialized.

LUA_RESOURCE_NOT_AVAILABLE

Secondary return code; the logical unit, physical unit, link, or link station specified in the request unit is unavailable. This return code is posted to **SLI_OPEN** when a resource is unavailable unless you use the retry option.

To use the retry option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue SLI_CLOSE.

LUA_SESSION_LIMIT_EXCEEDED

Secondary return code; the session requested was not activated because an NAU is at its session limit. This SNA sense code applies to the following requests: BID, CINIT, INIT, and ACTDRM.

The code will be posted to **SLI_OPEN** when an NAU is at its limit, unless you use the RETRY option.

To use the reset option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue **SLI_CLOSE**.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_NEGOTIABLE_BIND_ERROR

Secondary return code; a negotiable BIND was received, which is only allowed by the SLI when a user-supplied SLI_BIND routine is provided with **SLI_OPEN**.

LUA_BIND_FM_PROFILE_ERROR

Secondary return code; only file management header profiles 3 and 4 are supported by the LUA interface. A file management profile other than 3 or 4 was found on the BIND.

LUA_BIND_TS_PROFILE_ERROR

Secondary return code; only transmission service (TS) profiles 3 and 4 are supported by the LUA interface. A TS other than 3 or 4 was found on the BIND.

LUA_BIND_LU_TYPE_ERROR

Secondary return code; only LU 0, LU 1, LU 2, and LU 3 are supported by LUA. An LU other than 0, 1, 2, or 3 was found.

LUA_SSCP_LU_SESSION_NOT_ACTIVE

Secondary return code; the required SSCP-LU is inactive. Specific sense code information is in bytes 2 and 3. Valid settings are 0x0000, 0x0001, 0x0002, 0x0003, and 0x0004.

LUA_SESSION_SERVICES_PATH_ERROR

Secondary return code; a request for session services cannot be rerouted to an SSCP-SSCP session path. Specific sense code information in bytes 2 and 3 gives more information about why the request cannot be rerouted.

LUA_UNSUCCESSFUL

Primary return code; the verb record supplied was valid but the verb did not complete successfully.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_SESSION_ALREADY_OPEN

Secondary return code; a session is already open for the LU name specified in **SLI_OPEN**.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another process owns the session.

LUA_LINK_NOT_STARTED

Secondary return code; the LUA was not able to activate the data link during initialization of the session.

LUA_INVALID_ADAPTER

Secondary return code; the configuration for the data link control (DLC) is in error, or the configuration file is corrupted.

LUA_ENCR_DECR_LOAD_ERROR

Secondary return code; an unexpected return code was received from the OS/2 **DosLoadModule** function while attempting to load the user-provided encryption or decryption dynamic link module.

LUA_ENCR_DECR_PROC_ERROR

Secondary return code; an unexpected return code was received from the OS/2 **DosGetProcAddr** function while attempting to get the procedure address within the user-provided encryption or decryption dynamic link module.

LUA_NEG_NOTIFY_RSP

Secondary return code; the SSCP responded negatively to a NOTIFY request issued indicating that the secondary LU was capable of a session. The half-session component that received the request understood and supported the request but could not execute it.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the SLI was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_CANCELED

Primary return code; the secondary return code gives the reason for canceling the command.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process was canceled.

LUA_IN_PROGRESS

Primary return code; an asynchronous command was received but is not completed.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

For each **SLI_OPEN**, the Windows LUA interface:

- Starts the communication session.
- Reads and verifies a BIND command from the host, and passes it to the application if a BIND extension routine is supplied.
- Writes a BIND response.
- Reads and processes the STSN command and passes it to the application if a BIND extension is supplied (if necessary).
- Writes the STSN response (if necessary).
- Reads the CRV command (if necessary).
- Writes the CRV response (if necessary).
- Reads and processes the SDT command.
- Writes the SDT response.

The Windows LUA interface does the following additional functions for sessions that issue **SLI_OPEN** with the open type set to LUA_INIT_TYPE_SEC_IS or LUA_INIT_TYPE_SEC_LOG:

- Writes an INITSELF or an unformatted LOGON message.
- Reads and processes an INITSELF response or LOGON message response.

All SNA message traffic is administered by **SLI_OPEN** through the SDT command response.

To choose a certain LU configured for Windows LUA, the application sets lua_luname to the LU name in ASCII, padded with trailing spaces if necessary.

When SLI_OPEN is posted with LUA_OK in the lua_prim_rc parameter, SLI_OPEN successfully completed and the LU-LU data-flow session was established. The application can now issue SLI_BID, SLI_CLOSE, SLI_PURGE, SLI_RECEIVE, and SLI_SEND.

When SLI_OPEN is posted with a primary return code other than LUA_OK or LUA_IN_PROGRESS, the command did not successfully establish a session.

When using SLI_OPEN, a Windows LUA application must provide a session initialization type. Valid types are:

- Secondary with INITSELF
- Secondary with an Unformatted LOGON Message
- Primary Waiting for a BIND Command
- Primary with SSCP Access

See Also

RUI_INIT, SLI_OPEN, SLI_RECEIVE, SLI_SEND

Secondary with INITSELF

To initialize a session by having the secondary issue an INITSELF command, set **open.lua_init_type** to LUA_INIT_TYPE_SEC_IS. When this type of session initialization is chosen, the application has to format and provide the INITSELF command. The address of the INITSELF command is specified by **lua_data_ptr**. The actual length of the INITSELF command is specified by **lua_data_length**.

Secondary with an Unformatted LOGON Message

To initialize a session by having the secondary issue an unformatted LOGON message, set **open.lua_init_type** to LUA_INIT_TYPE_SEC_LOG. The length of the user's EBCDIC LOGON message is then specified in **lua_data_length**. The address of the user's EBCDIC LOGON message length is specified by **lua_data_ptr**.

Primary Waiting for a BIND Command

To initialize a session by having the secondary wait for the primary to issue a BIND and SDT, set **open.lua_init_type** to LUA_INIT_TYPE_PRIM. Until the host begins a session with the Windows LUA application using the BIND command followed by an SDT command, the **SLI_OPEN** issued stays IN_PROGRESS.

Primary with SSCP Access

To initialize a session by having the SLI wait for a BIND and SDT but allow SSCP access, set **open.lua_init_type** to LUA_INIT_TYPE_PRIM_SSCP. Rather than sending commands to the host to begin a session, the SLI allows the Windows LUA application to issue SLI_SEND and SLI_RECEIVE for the SSCP normal flow only. This allows the INITSELF commands or LOGON messages and responses to be transmitted between the Windows LUA application and the host. The application can have more than one INITSELF and LOGON message. For this type of session only, other SLI verbs can be issued before SLI_OPEN completes. When issuing **SLI_SEND**, an application should not specify any flow flag unless the application is sending a response, as specified in the **lua_message_type** parameter of **SLI_OPEN**. To obtain the INIT_COMPLETE status, the application must first issue **SLI_OPEN**, and then issue either SLI_BID or **SLI_RECEIVE**. The INIT_COMPLETE status notifies the application that the **SLI_SEND** and **SLI_RECEIVE** verbs for SSCP normal flow data can be issued.

BIND, CRV, and STSN Routines

For BIND and STSN routines supplied by the application, the names of DLLs and the entry points for procedures are passed in the SLI_OPEN verb's VCB. During **SLI_OPEN**, the BIND and STSN routines are called if the appropriate SNA request is received. When a BIND routine is not supplied by the application, the SLI performs a minimal check of the BIND commands and responds as necessary. If no STSN routine is supplied and an STSN request arrives, a positive response is issued by the SLI. If a CRV request arrives, a negative response is issued by the SLI.

Names for BIND and STSN routines are provided as extensions of the SLI_OPEN verb's VCB. The lua_extension_list_offset parameter provides the offset from the start of the VCB to the first name in the extension list.

The function prototype for a user-defined BIND or STSN routine on Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 is as follows:

```
VOID WINAPI UserFunction(
LUA_VERB_RECORD *lpVcb
);
```

The function prototype for a user-defined BIND, STSN, or END routine on Windows 3.x is as follows:

```
VOID FAR PASCAL UserFunction(
LUA_VERB_RECORD FAR *lpVcb
);
```

In both prototypes, the *lpVcb* parameter is a pointer to a LUA VCB.

BIND Example

The following example illustrates checking the incoming BIND image using these features of SLI_OPEN.

```
lua_vcb.specific.open.lua_open_extension[0].lua_routine_type  =
    LUA_ROUTINE_TYPE_BIND;
strcpy(lua_vcb.specific.open.lua_open_extension[0].lua_module_name,
    "WINSLI32");
strcpy(lua_vcb.specific.open.lua_open_extension[0].lua_procedure_name,
    "BindValidation");
lua_vcb.specific.open.lua_open_extension[1].lua_routine_type =
     LUA_ROUTINE_TYPE_END;
```

The following is the skeleton of a sample bind validation callback code for Windows 3.x:

```
VOID FAR PASCAL BindValidation ( LUA_VERB_RECORD FAR * pVerb )
 {
    pVerb->common.lua_prim_rc = LUA_STATE_CHECK;
 }
```

Note that for Microsoft Visual C++ 4.0 or later and Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95, the function prototype should be:

```
VOID WINAPI BindValidation (LUA_VERB_RECORD FAR * pVerb );
```

On Windows 2000, Windows NT, Windows 98, or Windows 95, the WINAPI macro equates to _STDCall.

The BIND routine has access to the LUA VCB passed to it. The BIND routine should validate the BIND and indicate the appropriate SLI primary and secondary return code in the LUA verb record. Also, the routine may indicate the primary and secondary RU sizes supported by the SLI program by setting bytes 10 and 11 in the common.lua_data_ptr field (where the BIND command is indicated).

The following are the Visual C++ compiler options for the module containing the callback:

```
 /FA -c -Zle -W3 -WX -Ge -Gy -Gz -Ox -Zd
 -DCONDITION_HANDLING -DSTD_CALL
 -Di386=1 -D_X86_ -DNT_UP -DWIN32 -DDEVL
 -D_DLL -D_MT  -DWIN32_SUPPORT
```

The following is the code generated for the callback:

```
PUBLIC _BindValidation@4
; COMDAT _BindValidation@4
_TEXT SEGMENT
   _pVerb$ = 8
   _BindValidation@4 PROC NEAR    ; COMDAT

   // pVerb->common.lua_prim_rc = LUA_STATE_CHECK;
   mov eax, DWORD PTR _pVerb$[esp-4]
   mov WORD PTR [eax+4], 512   ; 00000200H
   ret 4
_BindValidation@4 ENDP
_TEXT ENDS
```

The following is the code generated by SLI to call this callback:

```
   // (*aSCB->bind_rtn)(sliVCB);
```

```
    push    ebp
    call    DWORD PTR [ebx+188]
    // note there is no ADD ESP,4 following the call
```

The following is the client internal trace showing WINSLI detecting the user provided bind validation callback:

```
|00000157.000000f7 OUDMD  Opening User DLL Modules
|00000157.000000f7 OUDMD  Opening a Bind Routine
|00000157.000000f7 OUDMD  Opening DLL = WINSLI32
|00000157.000000f7 OUDMD  Loading Routine = BindValidation
```

The following is client internal trace showing the bind validation callback:

```
|00000157.0000015c CLUAD  Calling BIND Routine
|00000157.0000015c CLUAD  Return from BIND routine, prc=512
|00000157.0000015c CLUAD  Returned With Error From Routine
|00000157.0000015c FrRUI  Freeing RUI vcb = 0x14E424
|00000157.0000015c BINDP  USER BIND ROUTINE FAILED
```

The following is an API trace to show the bind validation error:

```
000015c SLI    ------------------------------------------   11:11:52.28
000015c SLI    SLI_OPEN post
000015c SLI    SESSION_FAILURE - NEG_RSP_FROM_BIND_ROUTINE
000015c SLI    ---- Verb Parameter Block at address 00405150 ----
000015c SLI    52004900 000F0000 00000039 01000000
    <R.I........9....>
000015c SLI    00000000 4C553220 20202020 48000000
    <....LU2     H...>
000015c SLI    88E01400 00000400 C0904000 F4000000
    <h.........@.4...>
000015c SLI    00000000 00000000 00000040 00000000
    <...........@....>
000015c SLI    00000000 02000000 0157494E 534C4933
    <.........WINSLI3>
000015c SLI    32004269 6E645661 6C696461 74696F6E
    <2.BindValidation>
000015c SLI    00000000 00000000 00000000 00000000
    <................>
000015c SLI    00000000 00000000 00000000 00000000
    <................>
000015c SLI    00000000 00000000 00000000 00000000
    <................>
000015c SLI    00000000 00000000 00000000 00000000
    <................>
000015c SLI    00000000 00000000 00000000 00000000
    <................>
000015c SLI    00000000 00000000 0000
    <..........      >
000015c SLI    ---- Data at address 004090C0 ----
000015c SLI    86998584
    <fred            >
```

Recovering from SESSION_FAILURE

If the SLI_OPEN completes with the primary return code of SESSION_FAILURE, the Windows LUA interface allows you to reissue **SLI_OPEN** without issuing SLI_CLOSE.

Ending a Pending SLI_OPEN

To end a pending SLI_OPEN, issue SLI_CLOSE with **lua_flag2.close_abend** set to ON.

SLI_PURGE

The **SLI_PURGE** verb cancels SLI_RECEIVE verbs issued with a wait condition.

The following structure describes the LUA_COMMON member of the VCB used by SLI_PURGE.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_PURGE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_PURGE only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_PURGE** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_PURGE** and should be set to zero.

lua_data_length

Not used by **SLI_PURGE** and should be set to zero.

lua_data_ptr
When **SLI_PURGE** is issued, this parameter points to the location of the SLI_RECEIVE verb's VCB that is to be canceled.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Not used by **SLI_PURGE** and should be set to zero.

lua_rh
Not used by **SLI_PURGE** and should be set to zero.

lua_flag1
Not used by **SLI_PURGE** and should be set to zero.

lua_message_type
Not used by **SLI_PURGE** and should be set to zero.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Not used by **SLI_PURGE** and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_NO_RECEIVE_TO_PURGE

Secondary return code; no SLI_RECEIVE was outstanding when you issued **SLI_PURGE**. One of two situations caused the problem:

- **SLI_RECEIVE** completed before **SLI_PURGE** finished processing. You can change the application to take care of this problem because it is not an error condition.
- The lua_data_ptr parameter does not correctly point to the SLI_RECEIVE you want to purge.

LUA_SLI_PURGE_PENDING

Secondary return code; an **SLI_PURGE** was still active when another **SLI_PURGE** was issued. Only one **SLI_PURGE** can be active at a time.

LUA_SESSION_FAILURE
Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_UNSUCCESSFUL
Primary return code; the verb record supplied was valid but the verb did not complete successfully.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_NOT_READY

Secondary return code; one of the following caused the SLI session to be temporarily suspended:

- An SNA UNBIND type 0x02 command was received, which indicates a new BIND is coming. If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while the **SLI_OPEN** is still processing, the primary return code is SESSION_FAILURE, not LUA_STATUS.
- The receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_INVALID_PROCESS

Secondary return code; the session for which an RUI verb was issued is unavailable because another OS/2 process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the RUI was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_CANCELED
Primary return code; the secondary return code gives the reason for canceling the command.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process was canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_PURGE cancels SLI_RECEIVE commands with a wait condition.

Typically, SLI_PURGE is issued if SLI_RECEIVE takes too long to complete. To cancel an SLI_RECEIVE, lua_data_ptr has to point to the SLI_RECEIVE VCB to cancel. The primary return code of the SLI_RECEIVE will be set to LUA_CANCELED when SLI_PURGE succeeds in canceling SLI_RECEIVE.

See Also

RUI_INIT, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

SLI_RECEIVE

The **SLI_RECEIVE** verb receives responses, SNA commands, and data into a Windows LUA application's buffer. **SLI_RECEIVE** also provides the current status of the session to the Windows LUA application.

The following structure describes the LUA_COMMON member of the VCB used by SLI_RECEIVE.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_RECEIVE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_RECEIVE only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_RECEIVE** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Specifies the length of received buffer for RUI_READ and **SLI_RECEIVE**.

lua_data_length
Returned parameter. Specifies the length of data returned in the receive buffer.

lua_data_ptr
Pointer to the application-supplied buffer that is to receive the data from an **SLI_RECEIVE** verb. Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

When SLI_RECEIVE is issued, this parameter points to the location to receive the data from the host.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x0 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. This parameter is used by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, **SLI_RECEIVE**, and SLI_SEND. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set **lua_flag1.bid_enable** to 1 to re-enable the most recent SLI_BID (equivalent to issuing **SLI_BID** again with exactly the same parameters as before), or set it to zero if you do not want to re-enable **SLI_BID**. Note that re-enabling the previous **SLI_BID** reuses the VCB originally allocated for it, so this VCB must not have been freed or modified.

Set lua_flag1.nowait to 1 to indicate that you want SLI_RECEIVE to return immediately whether or not data is available to be read, or set it to zero if you want the verb to wait for data before returning.

Set one or more of the following flags to 1 to indicate from which message flow to read data:

lua_flag1.sscp_exp

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

If more than one flag is set, the highest-priority data available is returned. The order of priorities (highest first) is: SSCP expedited, LU expedited, SSCP normal, LU normal. The equivalent flag in the **lua_flag2** group is set to indicate from which flow the data was read.

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. Returned parameter. Specifies the type of SNA message indicated to **SLI_RECEIVE**. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_STSN

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU-DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. Returned by RUI_BID, RUI_READ, **RUI_WRITE**, **SLI_BID**, **SLI_RECEIVE**, and SLI_SEND. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that **RUI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Not used by **SLI_RECEIVE** and should be set to zero.

lua_encr_decr_option
Not used by **SLI_RECEIVE** and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_BID_VERB_SEGMENT_ERROR

Secondary return code; the buffer with the SLI_BID VCB was released before the **SLI_RECEIVE** with **lua_flag1.bid_enable** set to 1 was issued.

LUA_NO_PREVIOUS_BID_ENABLED

Secondary return code; **SLI_BID** was not issued prior to issuing **SLI_RECEIVE** with **lua_flag1.bid_enable**.

LUA_BID_ALREADY_ENABLED

Secondary return code; **SLI_RECEIVE** was issued with **lua_flag1.bid_enable** when **SLI_BID** was already active.

LUA_INVALID_FLOW

Secondary return code; the **lua_flag1** flow flags were set incorrectly when a verb was issued:

- When issuing SLI_SEND to send an SNA response, set only one **lua_flag1** flow flag.
- When issuing SLI_RECEIVE, set at least one lua_flag1 flow flag.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_RECEIVE_ON_FLOW_PENDING

Secondary return code; an **SLI_RECEIVE** was still outstanding when this application issued another **SLI_RECEIVE** for an SNA flow.

LUA_SESSION_FAILURE

Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RUI_WRITE_FAILURE

Secondary return code; an unexpected error was posted to the SLI by RUI_WRITE.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_RECEIVER_IN_TRANSMIT_MODE

Secondary return code; either resources needed to handle normal flow data were not available or the state of the half-duplex contention was not received when a normal-flow request was received. The result is a race condition. This SNA sense code is also an exception request sense code.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_CHAINING_ERROR

Secondary return code; the sequence of the chain indicator settings is in error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_BRACKET

Secondary return code; the sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DIRECTION

Secondary return code; while the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_QUIESCED

Secondary return code; a DFC or FMD request was received from a half-session that sent either a SHUTC command or QC command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_NO_BEGIN_BRACKET

Secondary return code; the receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_IMMEDIATE_REQUEST_MODE_ERROR

Secondary return code; the request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_QUEUED_RESPONSE_ERROR

Secondary return code; the request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_ERP_SYNC_EVENT_ERROR

Secondary return code; a violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_CORRELATION_ERROR

Secondary return code; a response was sent that does not correspond to a previously received request or a response was received that does not correspond to a previously sent request.

LUA_RSP_PROTOCOL_ERROR

Secondary return code; a violation of the response protocol was found in the response received from the primary half-session.

LUA_BB_NOT_ALLOWED

Secondary return code; the begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EB_NOT ALLOWED

Secondary return code; the end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EXCEPTION_RSP_NOT_ALLOWED

Secondary return code; when an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_DEFINITE_RSP_NOT_ALLOWED

Secondary return code; when a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_ALLOWED

Secondary return code; the change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_RESPONSE_NOT_ALLOWED

Secondary return code; a request other than an EXR contained a NO RESPONSE. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CHAINING_NOT_SUPPORTED

Secondary return code; the chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_BRACKETS_NOT_SUPPORTED

Secondary return code; the bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_SUPPORTED

Secondary return code; the change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_FI

Secondary return code; the format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

Secondary return code; the code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_RU_CATEGORY

Secondary return code; the request unit category indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_REQUEST_CODE

Secondary return code; the request code was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_SPEC_OF_SDI_RTI

Secondary return code; the SDI and the RTI were not specified correctly on a response. The BIND options chosen previously or

the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_DR1I_DR2I_ERI

Secondary return code; the DR1I, the DR2I, and the ERI were specified incorrectly. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_QRI

Secondary return code; the queued response indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_EDI

Secondary return code; the EDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_PDI

Secondary return code; the PDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_UNSUCCESSFUL

Primary return code; the verb record supplied was valid but the verb did not complete successfully.

LUA_DATA_TRUNCATED

Secondary return code; the data was truncated because the data received was longer than the buffer length specified in **lua_max_length**.

LUA_DATA_SEGMENT_LENGTH_ERROR

Secondary return code; one of the following has occurred:

- The supplied data segment for **SLI_RECEIVE** or SLI_SEND is not a read/write data segment as required.
- The supplied data segment for SLI_RECEIVE is not as long as that provided in lua_max_length.
- The supplied data segment for SLI_SEND is not as long as that provided in lua_data_length.

LUA_NO_DATA

Secondary return code; no data was available to read when **SLI_RECEIVE** containing a no wait parameter was issued.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_NOT_READY

Secondary return code; one of the following has caused the SLI session to be temporarily suspended:

- An SNA UNBIND type 0x02 command was received, which indicates a new BIND is coming. If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while the **SLI_OPEN** is still processing, the primary return code is LUA_SESSION_FAILURE, not LUA_STATUS.
- The receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another OS/2 process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the LUA was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_RECEIVE_CORRELATION_TABLE_FULL

Secondary return code; the session receive correlation table for the flow requested reached its capacity.

LUA_NEGATIVE_RESPONSE

Primary return code; either the LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; the LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_DATA_TRAFFIC_RESET

Secondary return code; a half-session of an active session but with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_NOT_RESET

Secondary return code; while the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_SC_PROTOCOL_VIOLATION

Secondary return code; a violation of SC protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_INVALID_SC_OR_NC_RH

Secondary return code; the RH of an SC or NC request was invalid.

LUA_PACING_NOT_SUPPORTED

Secondary return code; the request contained a pacing indicator when support of pacing for this session does not exist for the receiving half-session or boundary function half-session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NAU_INOPERATIVE

Secondary return code; the network addressable unit is not able to process responses or requests. Delivery to the receiver could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, it usually means there is no longer a valid path to the session partner.

LUA_CANCELED
Primary return code; the secondary return code gives the reason for canceling the command.

LUA_PURGED

Secondary return code; SLI_PURGE was issued and canceled **SLI_RECEIVE**.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_CANCEL_COMMAND_RECEIVED

Secondary return code; the host sent an SNA CANCEL command to cancel the data chain currently being received by **SLI_RECEIVE**.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process has been canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.
LUA_STATUS
Primary return code; the secondary return code contains SLI status information for the application.

LUA_READY

Secondary return code; following a NOT READY status, this status is issued to notify you that the SLI is ready to process commands.

LUA_NOT_READY

Secondary return code; the SLI session is temporarily suspended for the following reason:

- An SNA UNBIND type 0x02 command was received, which means a new BIND is coming. If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while the **SLI_OPEN** is still processing, the primary return code is session-failure, not status.
- The receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_INIT_COMPLETE

Secondary return code; the LUA interface initialized the session while SLI_OPEN was processing. LUA applications that issue **SLI_OPEN** with **lua_open_type_prim_sscp** receive this status on **SLI_RECEIVE** or SLI_BID.

LUA_SESSION_END_REQUESTED

Secondary return code; the LUA interface received an SNA SHUTD from the host, which means the host is ready to shut down the session.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.
LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.
LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.
LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_RECEIVE receives responses, SNA commands, and request unit data from the host. **SLI_RECEIVE** also provides the status of the session to the Windows LUA application. An SLI_OPEN request must complete before **SLI_RECEIVE** can be issued. However, if **SLI_OPEN** is issued with **lua_init_type** set to LUA_INIT_TYPE_PRIM_SSCP, an **SLI_RECEIVE** over the SSCP normal flow can be issued as soon as **SLI_OPEN** returns an IN_PROGRESS.

Data is received by the application in one of four session flows. The four session flows, from highest to lowest priority are:

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

The data flow type that **SLI_RECEIVE** will process is specified in **lua_flag1**. The application can also specify whether it wants to look at more than one type of data flow. When multiple flow bits are set, the highest priority is received first. When **SLI_RECEIVE** completes processing, **lua_flag2** indicates the specific type of flow for which data has been received by the Windows LUA application.

If SLI_BID successfully completes before SLI_RECEIVE is issued, the Windows LUA interface can be instructed to reuse the last SLI_BID verb's VCB. To do this, issue SLI_RECEIVE with lua_flag1.bid_enable set to 1.

When using lua_flag1.bid_enable, the SLI_BID storage must not be freed because the last SLI_BID verb's VCB is used. Also, when using lua_flag1.bid_enable, the successful completion of SLI_BID will be posted.

If SLI_RECEIVE is issued with lua_flag1.nowait when no data is available to receive, LUA_NO_DATA will be the secondary return code set by the Windows LUA interface.

Session Status Return Values

If LUA_STATUS is the primary return code, the secondary return code can be one of the following:

LUA_READY

LUA_NOT_READY

LUA_SESSION_END_REQUESTED

LUA_INIT_COMPLETE

In addition, if LUA_STATUS is the primary return code, the following parameters are used:

lua_sec_rc

lua_sid

LUA_READY is returned after an LUA_NOT_READY status and indicates that the SLI is again ready to perform all commands.

LUA_NOT_READY indicates that the SLI session is suspended because the SLI has received either an SNA CLEAR command or an SNA UNBIND command with an 0x02 UNBIND type (UNBIND with BIND forthcoming). Depending on what caused the suspension, the session can be reactivated as follows:

- When the suspension is caused by an SNA CLEAR, receiving an SNA SDT reactivates the session.
- When an SNA UNBIND type BIND forthcoming causes suspension of the session and the SLI_OPEN that opened the session is completed, the session is suspended until the SLI receives a BIND and SDT command. The session can also optionally receive an STSN command. As a result, user-supplied routines issued with the initial SLI_OPEN must be re-entered because they will be recalled.

The application can send SSCP data after a CLEAR or UNBIND type BIND forthcoming arrives and before the NOT_READY status is read. The application can send and receive SSCP data after reading a NOT_READY.

When an SNA UNBIND type BIND forthcoming arrives before completion of the SLI_OPEN that opened the session, LUA_SESSION_FAILURE (not LUA_STATUS) is the primary return code.

LUA_SESSION_END_REQUESTED indicates that the application received an SNA SHUTD from the host. The Windows LUA

application should issue SLI_CLOSE to close the session when convenient.

LUA_INIT_COMPLETE is returned only when lua_init_type for SLI_OPEN is LUA_INIT_TYPE_PRIM_SSCP. The status means that the SLI_OPEN has been processed sufficiently to allow SSCP data to now be sent or received.

Exception Requests

If a host application request unit is converted into an EXR, sense data will be returned. When SLI_BID completes with the returned verb parameters set as shown, an EXR conversion occurs.

Member	Set to
lua_prim_rc	OK (0x0000)
lua_sec_rc	OK (0x00000000)
lua_rh.rri	bit off (request unit)
lua_rh.sdi	bit on (includes sense data)

Of the seven bytes of data in **lua_peek_data**, bytes 0 through 3 define the error detected. The following table indicates possible sense data and the values of bytes 0 through 3.

Sense data	Value of bytes 0–3
LUA_MODE_INCONSISTENCY	0x08090000
LUA_BRACKET_RACE_ERROR	0x080B0000
LUA_BB_REJECT_NO_RTR	0x08130000
LUA_RECEIVER_IN_TRANSMIT_MODE	0x081B0000
LUA_CRYPTOGRAPHY_FUNCTION_INOP	0x08480000
LUA_SYNC_EVENT_RESPONSE	0x10010000
LUA_RU_DATA_ERROR	0x10020000
LUA_RU_LENGTH_ERROR	0x10020000
LUA_INCORRECT_SEQUENCE_NUMBER	0x20010000

The information returned to bytes 3 through 6 in **lua_peek_data** is determined by the first three bytes of the initial request unit that caused the error.

See Also

RUI_INIT, RUI_PURGE, RUI_READ, RUI_WRITE, SLI_BID, SLI_CLOSE, SLI_OPEN, SLI_PURGE, SLI_SEND

SLI_RECEIVE_EX

The **SLI_RECEIVE_EX** verb receives responses, SNA commands, and data into a Windows LUA application's buffer. **SLI_RECEIVE_EX** also provides the current status of the session to the Windows LUA application.

The SLI_RECEIVE_EX verb also supports inbound chaining. The maximum length of data that can be received by a single verb is 4,294,967,295 bytes. This is compared to a maximum of 65,535 bytes that can be received by the SLI_RECEIVE verb.

The following structure describes the LUA_COMMON member of the VCB used by SLI_RECEIVE_EX.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **SLI_RECEIVE_EX**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    struct SLI_RECEIVE_EX_SPECIFIC {
        unsigned long lua_data_length_ex;
        unsigned long lua_max_length_ex;
    };
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_RECEIVE_EX.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_RECEIVE_EX only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_RECEIVE_EX** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
This supplied parameter is reserved and must be set to zero.

The maximum length of data returned in a receive buffer must be set in the lua_max_length_ex parameter.

lua_data_length
This parameter is reserved and must be set to zero.

The length of data returned in the receive buffer is set in the lua_data_length_ex parameter.

lua_data_ptr
Pointer to the application-supplied buffer that is to receive the data from an **SLI_RECEIVE_EX** verb. Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

When SLI_RECEIVE_EX is issued, this parameter points to the location to receive the data from the host.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. This parameter is used by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, **SLI_RECEIVE_EX**, and SLI_SEND_EX. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set **lua_flag1.bid_enable** to 1 to re-enable the most recent SLI_BID (equivalent to issuing **SLI_BID** again with exactly the same parameters as before), or set it to zero if you do not want to re-enable **SLI_BID**. Note that re-enabling the previous **SLI_BID** reuses the VCB originally allocated for it, so this VCB must not have been freed or modified.

Set lua_flag1.nowait to 1 to indicate that you want SLI_RECEIVE_EX to return immediately whether or not data is available to be read, or set it to zero if you want the verb to wait for data before returning.

Set one or more of the following flags to 1 to indicate from which message flow to read data:

lua_flag1.sscp_exp

lua_flag1.lu_exp

lua_flag1.sscp_norm

lua_flag1.lu_norm

If more than one flag is set, the highest-priority data available is returned. The order of priorities (highest first) is: SSCP expedited, LU expedited, SSCP normal, LU normal. The equivalent flag in the **lua_flag2** group is set to indicate from which flow the data was read.

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. Returned parameter. Specifies the type of SNA message indicated to **SLI_RECEIVE_EX**. Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_STSN

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU-DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA. Returned by RUI_BID, RUI_READ, **RUI_WRITE**, **SLI_BID**, **SLI_RECEIVE**, and SLI_SEND_EX. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that **RUI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Not used by **SLI_RECEIVE** and should be set to zero.
lua_encr_decr_option
Not used by **SLI_RECEIVE** and should be set to zero.
lua_max_length_ex
Specifies the length of received buffer for **SLI_RECEIVE_EX**.
lua_data_length_ex
The union member of **LUA_SPECIFIC** used by **SLI_RECEIVE_EX**. Returned parameter. Specifies the length of data returned in the receive buffer.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write

segment and supplied data is required.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_BID_VERB_SEGMENT_ERROR

Secondary return code; the buffer with the SLI_BID VCB was released before the **SLI_RECEIVE_EX** with **lua_flag1.bid_enable** set to 1 was issued.

LUA_NO_PREVIOUS_BID_ENABLED

Secondary return code; **SLI_BID** was not issued prior to issuing **SLI_RECEIVE_EX** with **lua_flag1.bid_enable**.

LUA_BID_ALREADY_ENABLED

Secondary return code; **SLI_RECEIVE_EX** was issued with **lua_flag1.bid_enable** when **SLI_BID** was already active.

LUA_INVALID_FLOW

Secondary return code; the **lua_flag1** flow flags were set incorrectly when a verb was issued:

- When issuing SLI_SEND_EX_sna_SLI_SEND_EX_lua to send an SNA response, set only one lua_flag1 flow flag.
- When issuing SLI_RECEIVE, set at least one lua_flag1 flow flag.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_RECEIVE_ON_FLOW_PENDING

Secondary return code; an **SLI_RECEIVE_EX** was still outstanding when this application issued another **SLI_RECEIVE_EX** for an SNA flow.

LUA_SESSION_FAILURE

Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RUI_WRITE_FAILURE

Secondary return code; an unexpected error was posted to the SLI by RUI_WRITE.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_RECEIVER_IN_TRANSMIT_MODE

Secondary return code; either resources needed to handle normal flow data were not available or the state of the half-duplex contention was not received when a normal-flow request was received. The result is a race condition. This SNA sense code is also an exception request sense code.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_CHAINING_ERROR

Secondary return code; the sequence of the chain indicator settings is in error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_BRACKET

Secondary return code; the sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DIRECTION

Secondary return code; while the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_QUIESCED

Secondary return code; a DFC or FMD request was received from a half-session that sent either a SHUTC command or QC command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_NO_BEGIN_BRACKET

Secondary return code; the receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_IMMEDIATE_REQUEST_MODE_ERROR

Secondary return code; the request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_QUEUED_RESPONSE_ERROR

Secondary return code; the request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_ERP_SYNC_EVENT_ERROR

Secondary return code; a violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_CORRELATION_ERROR

Secondary return code; a response was sent that does not correspond to a previously received request or a response was received that does not correspond to a previously sent request.

LUA_RSP_PROTOCOL_ERROR

Secondary return code; a violation of the response protocol was found in the response received from the primary half-session.

LUA_BB_NOT_ALLOWED

Secondary return code; the begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EB_NOT ALLOWED

Secondary return code; the end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EXCEPTION_RSP_NOT_ALLOWED

Secondary return code; when an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_DEFINITE_RSP_NOT_ALLOWED

Secondary return code; when a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_ALLOWED

Secondary return code; the change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_RESPONSE_NOT_ALLOWED

Secondary return code; a request other than an EXR contained a NO RESPONSE. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CHAINING_NOT_SUPPORTED

Secondary return code; the chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_BRACKETS_NOT_SUPPORTED

Secondary return code; the bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_SUPPORTED

Secondary return code; the change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_FI

Secondary return code; the format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors

are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

Secondary return code; the code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_RU_CATEGORY

Secondary return code; the request unit category indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_REQUEST_CODE

Secondary return code; the request code was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_SPEC_OF_SDI_RTI

Secondary return code; the SDI and the RTI were not specified correctly on a response. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_DR1I_DR2I_ERI

Secondary return code; the DR1I, the DR2I, and the ERI were specified incorrectly. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_QRI

Secondary return code; the queued response indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_EDI

Secondary return code; the EDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_PDI

Secondary return code; the PDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_UNSUCCESSFUL

Primary return code; the verb record supplied was valid but the verb did not complete successfully.

LUA_DATA_TRUNCATED

Secondary return code; the data was truncated because the data received was longer than the buffer length specified in **lua_max_length_ex**.

LUA_DATA_SEGMENT_LENGTH_ERROR

Secondary return code; one of the following has occurred:

- The supplied data segment for **SLI_RECEIVE_EX** or SLI_SEND_EX is not a read/write data segment as required.
- The supplied data segment for SLI_RECEIVE_EX is not as long as that provided in lua_max_length_ex.
- The supplied data segment for SLI_SEND_EX is not as long as that provided in lua_data_length_ex.

LUA_NO_DATA

Secondary return code; no data was available to read when **SLI_RECEIVE_EX** containing a no wait parameter was issued.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_NOT_READY

Secondary return code; one of the following has caused the SLI session to be temporarily suspended:

- An SNA UNBIND type 0x02 command was received, which indicates a new BIND is coming. If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while the **SLI_OPEN** is still processing, the primary return code is LUA_SESSION_FAILURE, not LUA_STATUS.
- The receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the LUA was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_RECEIVE_CORRELATION_TABLE_FULL

Secondary return code; the session receive correlation table for the flow requested reached its capacity.

LUA_NEGATIVE_RESPONSE

Primary return code; either the LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; the LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_DATA_TRAFFIC_RESET

Secondary return code; a half-session of an active session but with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_NOT_RESET

Secondary return code; while the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_SC_PROTOCOL_VIOLATION

Secondary return code; a violation of SC protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or request unit for the

received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_INVALID_SC_OR_NC_RH

Secondary return code; the RH of an SC or NC request was invalid.

LUA_PACING_NOT_SUPPORTED

Secondary return code; the request contained a pacing indicator when support of pacing for this session does not exist for the receiving half-session or boundary function half-session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NAU_INOPERATIVE

Secondary return code; the network addressable unit is not able to process responses or requests. Delivery to the receiver could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, it usually means there is no longer a valid path to the session partner.

LUA_CANCELED
Primary return code; the secondary return code gives the reason for canceling the command.

LUA_PURGED

Secondary return code; SLI_PURGE was issued and canceled **SLI_RECEIVE**.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_CANCEL_COMMAND_RECEIVED

Secondary return code; the host sent an SNA CANCEL command to cancel the data chain currently being received by **SLI_RECEIVE_EX**.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process has been canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.
LUA_STATUS
Primary return code; the secondary return code contains SLI status information for the application.

LUA_READY

Secondary return code; following a NOT READY status, this status is issued to notify you that the SLI is ready to process commands.

LUA_NOT_READY

Secondary return code; the SLI session is temporarily suspended for the following reason:

- An SNA UNBIND type 0x02 command was received, which means a new BIND is coming. If the UNBIND type 0x02 is received after the beginning SLI_OPEN is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while the **SLI_OPEN** is still processing, the primary return code is session-failure, not status.
- The receipt of an SNA CLEAR caused the suspension. Receipt of an SNA SDT will cause the session to resume.

LUA_INIT_COMPLETE

Secondary return code; the LUA interface initialized the session while SLI_OPEN was processing. LUA applications that issue **SLI_OPEN** with **lua_open_type_prim_sscp** receive this status on **SLI_RECEIVE** or SLI_BID.

LUA_SESSION_END_REQUESTED

Secondary return code; the LUA interface received an SNA SHUTD from the host, which means the host is ready to shut down the session.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_RECEIVE_EX receives responses, SNA commands, and request unit data from the host. **SLI_RECEIVE_EX** also provides the status of the session to the Windows LUA application.

The difference between SLI_RECEIVE_EX and SLI_RECEIVE is that the SLI_RECEIVE_EX verb supports inbound chaining and can receive up to 4,295 KB in a single verb request. In contrast, SLI_RECEIVE is limited to receiving up to 64 KB in a verb request.

An SLI_OPEN request must complete before SLI_RECEIVE_EX can be issued. However, if SLI_OPEN is issued with lua_init_type set to LUA_INIT_TYPE_PRIM_SSCP, an SLI_RECEIVE_EX over the SSCP normal flow can be issued as soon as SLI_OPEN returns an IN_PROGRESS.

Data is received by the application in one of four session flows. The four session flows, from highest to lowest priority are:

- SSCP expedited
- LU expedited
- SSCP normal
- LU normal

The data flow type that **SLI_RECEIVE_EX** will process is specified in **lua_flag1**. The application can also specify whether it wants to look at more than one type of data flow. When multiple flow bits are set, the highest priority is received first. When **SLI_RECEIVE_EX** completes processing, **lua_flag2** indicates the specific type of flow for which data has been received by the Windows LUA application.

If SLI_BID successfully completes before SLI_RECEIVE is issued, the Windows LUA interface can be instructed to reuse the last SLI_BID verb's VCB. To do this, issue SLI_RECEIVE_EX with lua_flag1.bid_enable set to 1.

When using lua_flag1.bid_enable, the SLI_BID storage must not be freed because the last SLI_BID verb's VCB is used. Also, when using lua_flag1.bid_enable, the successful completion of SLI_BID will be posted.

If SLI_RECEIVE_EX is issued with lua_flag1.nowait when no data is available to receive, LUA_NO_DATA will be the secondary return code set by the Windows LUA interface.

Session Status Return Values

If LUA_STATUS is the primary return code, the secondary return code can be one of the following:

LUA_READY

LUA_NOT_READY

LUA_SESSION_END_REQUESTED

LUA_INIT_COMPLETE

In addition, if LUA_STATUS is the primary return code, the following parameters are used:

lua_sec_rc

lua_sid

LUA_READY is returned after an LUA_NOT_READY status and indicates that the SLI is again ready to perform all commands.

LUA_NOT_READY indicates that the SLI session is suspended because the SLI has received either an SNA CLEAR command or an SNA UNBIND command with an 0x02 UNBIND type (UNBIND with BIND forthcoming). Depending on what caused the suspension, the session can be reactivated as follows:

- When the suspension is caused by an SNA CLEAR, receiving an SNA SDT reactivates the session.
- When an SNA UNBIND type BIND forthcoming causes suspension of the session and the SLI_OPEN that opened the session is completed, the session is suspended until the SLI receives a BIND and SDT command. The session can also optionally receive an STSN command. As a result, user-supplied routines issued with the initial SLI_OPEN must be re-entered because they will be recalled.

The application can send SSCP data after a CLEAR or UNBIND type BIND forthcoming arrives and before the NOT_READY status is read. The application can send and receive SSCP data after reading a NOT_READY.

When an SNA UNBIND type BIND forthcoming arrives before completion of the SLI_OPEN that opened the session, LUA_SESSION_FAILURE (not LUA_STATUS) is the primary return code.

LUA_SESSION_END_REQUESTED indicates that the application received an SNA SHUTD from the host. The Windows LUA application should issue SLI_CLOSE to close the session when convenient.

LUA_INIT_COMPLETE is returned only when lua_init_type for SLI_OPEN is LUA_INIT_TYPE_PRIM_SSCP. The status means that the SLI_OPEN has been processed sufficiently to allow SSCP data to now be sent or received.

Exception Requests

If a host application request unit is converted into an EXR, sense data will be returned. When SLI_BID completes with the returned verb parameters set as shown, an EXR conversion occurs.

Member	**Set to**
lua_prim_rc	OK (0x0000)
lua_sec_rc	OK (0x00000000)
lua_rh.rri	bit off (request unit)
lua_rh.sdi	bit on (includes sense data)

Of the seven bytes of data in **lua_peek_data**, bytes 0 through 3 define the error detected. The following table indicates possible sense data and the values of bytes 0 through 3.

Sense data	**Value of bytes 0–3**
LUA_MODE_INCONSISTENCY	0x08090000
LUA_BRACKET_RACE_ERROR	0x080B0000
LUA_BB_REJECT_NO_RTR	0x08130000
LUA_RECEIVER_IN_TRANSMIT_MODE	0x081B0000
LUA_CRYPTOGRAPHY_FUNCTION_INOP	0x08480000
LUA_SYNC_EVENT_RESPONSE	0x10010000
LUA_RU_DATA_ERROR	0x10020000
LUA_RU_LENGTH_ERROR	0x10020000
LUA_INCORRECT_SEQUENCE_NUMBER	0x20010000

The information returned to bytes 3 through 6 in **lua_peek_data** is determined by the first three bytes of the initial request unit that caused the error.

See Also

RUI_INIT, RUI_PURGE, RUI_READ, RUI_WRITE, SLI_BID, SLI_CLOSE, SLI_OPEN, SLI_PURGE, SLI_SEND_EX

SLI_SEND

The **SLI_SEND** verb sends responses, SNA commands, and data from a Windows LUA application to a host LU.

The following structure describes the LUA_COMMON member of the VCB used by SLI_SEND.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **SLI_SEND**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    unsigned char lua_sequence_number[2];
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_SEND.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_SEND only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_SEND** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_SEND** and should be set to zero.

lua_data_length
Supplied parameter. Specifies the length of data being sent.

lua_data_ptr
Pointer to the application-supplied buffer that contains the data to be sent to the host by **SLI_SEND**.

Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are set for write functions and returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Supplied parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for RUI_WRITE and **SLI_SEND**, and returned by RUI_READ and RUI_BID. For the RH for **SLI_SEND**, all fields except the queued-response indicator (**lua_rh.qri**) and pacing indicator (**lua_rh.pi**) are used.

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits.

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application.Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set one of the following flags to 1 to indicate on which message flow the data is to be sent:

lua_flag1.sscp_exp

lua_flag1.sscp_norm

lua_flag1.lu_exp

lua_flag1.lu_norm

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. This is a supplied parameter for **SLI_SEND**.

Possible values are as follows:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU-DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that RUI_BID was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.
lua_encr_decr_option
Not used by **SLI_SEND** and should be set to zero.
lua_sequence_number
The union member of **LUA_SPECIFIC** used by **SLI_SEND**. Returned parameter. Contains the sequence number for either the first in the chain request unit or the only segment in the chain request unit. Note that this parameter is not byte-reversed.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_INVALID_FLOW

Secondary return code; the **lua_flag1** flow flags were set incorrectly when a verb was issued:

- When issuing **SLI_SEND** to send an SNA response, set only one **lua_flag1** flow flag.
- When issuing SLI_RECEIVE, set at least one lua_flag1 flow flag.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_REQUIRED_FIELD_MISSING

Secondary return code; the verb that was issued either did not include a data pointer (if the data count was not zero) or did not include an **lua_flag1** flow flag.

LUA_INVALID_MESSAGE_TYPE

Secondary return code; the **lua_message_type** parameter is not recognized by the LUA interface.

LUA_DATA_LENGTH_ERROR

Secondary return code; the application did not provide user-supplied data required by the verb issued. Note that when **SLI_SEND** is issued for an SNA LUSTAT command, status (in four bytes) is required, and that when SLI_OPEN is issued with secondary initialization, data is required.

LUA_STATE_CHECK

Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_MAX_NUMBER_OF_SENDS

Secondary return code; the application issued a third **SLI_SEND** before one completed.

LUA_SEND_ON_FLOW_PENDING

Secondary return code; an **SLI_SEND** was still outstanding when the application issued another **SLI_SEND** for an SNA flow.

LUA_SESSION_FAILURE

Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued, or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_DATA_SEGMENT_LENGTH_ERROR

Secondary return code; one of the following has occurred:

- The supplied data segment for SLI_RECEIVE or **SLI_SEND** is not a read/write data segment as required.
- The supplied data segment for SLI_RECEIVE is not as long as that provided in lua_max_length.
- The supplied data segment for SLI_SEND is not as long as that provided in lua_data_length.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another OS/2 process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the LUA was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_INSUFFICIENT_RESOURCES

Secondary return code; a temporary condition of insufficient resources caused the request receiver to be unable to perform. The request sent to the half-session component was not executed, even though it was understood and supported.

LUA_SEND_CORRELATION_TABLE_FULL

Secondary return code; the session send correlation table for the flow requested reached its capacity.

LUA_RU_LENGTH_ERROR

Secondary return code; the RU request was an incorrect length (either too short or too long). The request unit was not interpreted or processed even though it was delivered to the half-session component. The half-session capabilities do not match. This SNA sense code is also an exception request sense code.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_HDX_BRACKET_STATE_ERROR

Secondary return code; the existing state error prevented the current request from being sent. The determination was made by a protocol computer.

LUA_RESPONSE_ALREADY_SENT

Secondary return code; a response for the chain was already sent so that the current request was not sent. The determination was made by a protocol computer.

LUA_EXR_SENSE_INCORRECT

Secondary return code; the application responded negatively to an exception request. The sense code was unacceptable.

LUA_RESPONSE_OUT_OF_ORDER

Secondary return code; the current response was not for the oldest request. The determination was made by a protocol computer.

LUA_CHAIN_RESPONSE_REQUIRED

Secondary return code; a CHASE response was still outstanding when a more recent request was attempted. The determination was made by a protocol computer.

LUA_BRACKET

Secondary return code; the sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DIRECTION

Secondary return code; while the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_RESET

Secondary return code; a half-session of an active session but with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_QUIESCED

Secondary return code; a DFC or FMD request was received from a half-session that sent either a SHUTC command or QC

command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_NOT_RESET

Secondary return code; while the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_NO_BEGIN_BRACKET

Secondary return code; the receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_SC_PROTOCOL_VIOLATION

Secondary return code; a violation of SC protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or data flow control state was found. Delivery to the half-session component was prevented.

LUA_IMMEDIATE_REQUEST_MODE_ERROR

Secondary return code; the request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_QUEUED_RESPONSE_ERROR

Secondary return code; the request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_ERP_SYNC_EVENT_ERROR

Secondary return code; a violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_BEFORE_SENDING_REQ

Secondary return code; a previously received request has not been responded to yet and an attempt was made in half-duplex send/receive mode to send a normal flow request. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_CORRELATION_ERROR

Secondary return code; a response was sent that does not correspond to a previously received request or a response was received that does not correspond to a previously sent request.

LUA_BB_NOT_ALLOWED

Secondary return code; the begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EB_NOT ALLOWED

Secondary return code; the end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EXCEPTION_RSP_NOT_ALLOWED

Secondary return code; when an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_DEFINITE_RSP_NOT_ALLOWED

Secondary return code; when a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_ALLOWED

Secondary return code; the change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_RESPONSE_NOT_ALLOWED

Secondary return code; a request other than an EXR contained a "no response." The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CHAINING_NOT_SUPPORTED

Secondary return code; the chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_BRACKETS_NOT_SUPPORTED

Secondary return code; the bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_SUPPORTED

Secondary return code; the change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_FI

Secondary return code; the format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

Secondary return code; the code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_REQUEST_CODE

Secondary return code; the request code was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_SPEC_OF_SDI_RTI

Secondary return code; the SDI and the RTI were not specified correctly on a response. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_DR1I_DR2I_ERI

Secondary return code; the DR1I, the DR2I, and the ERI were specified incorrectly. The BIND options chosen previously or the

architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_QRI

Secondary return code; the queued response indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_EDI

Secondary return code; the EDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_PDI

Secondary return code; the PDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_SESSION

Secondary return code; a request to activate a session is required because no active half-session in the receiving end node for the origination-destination pair exists, or no active boundary function half-session component for the origination-destination pair in a node that supplies the boundary function exists. Delivery of the request could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, that usually indicates there is no longer a valid path to the session partner.

LUA_CANCELED
Primary return code; the secondary return code gives the reason for canceling the command.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process has been canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT
Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR
Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL
Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB
Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_SEND sends responses, SNA commands, and data from the Windows LUA application to a host LU. A session must already be

open to issue **SLI_SEND** for a particular LU-LU session flow. To send data on the SSCP normal flow prior to the completion of SLI_OPEN, the session must have been initialized as primary with SSCP access. In addition, the session status must be INIT_COMPLETE.

The settings for lua_message_type determine the type of processing that will be done by SLI_SEND. The following table indicates the parameters to set based on the value of lua_message_type.

SLI_SEND parameter	LU_DATA SSCP_DATA	BID BIS RTR	CHASE QC	LUSTAT_LU LUSTAT_SSCP	QEC RELQ SBI SIGNAL	RQR	RSP
lua_data_length	Req.	0	0	Req.	0	0	Req. (0 if no data)
lua_data_ptr	Req. (0 if no data)	0	0	Req.	0	0	Req. (0 if no data)
lua_flag1 flow flags	0	0	0	0	0	0	Req. (set one)
lua_rh	FI DRL1 DRL2 RI BBI EBI CDI CSI EDI	SDI QRI	SDI QRI EBI CDI	SDI QRI DRL1 DRL2 RI BBI EBI CDI	SDI	0	RRI RI
lua_th	0	0	0	0	0	0	SNF

The location provided in **lua_data_ptr** and the length provided in **lua_data_length** determine the data that the SLI sends. The data will be chained by the SLI verbs if necessary.

When sending a response, the type of response determines the SLI_SEND information required. For all responses, you must:

- Set the selected **lua_flag1** flow flag.
- Provide the sequence number in lua_th.snf for the request to which you are responding.
- Set lua_message_type to LUA_MESSAGE_TYPE_RSP.

For multichain message responses, the sequence number of the last received chain element must be used. For a response to a multichain message ending with a CANCEL command, the CANCEL command sequence number is used.

For positive responses that only require the request code, set lua_rh.ri to zero (indicating that the response is positive) and lua_data_length to zero (indicating no data is provided). The request code is filled in by the SLI, using the sequence number provided.

For negative responses in which lua_rh.ri is set to 1, set the lua_data_ptr to the SNA sense code address and the lua_data_length to the SNA sense code length (four bytes). The sequence number is used by the SLI to fill in the request code.

See Also

RUI_INIT, RUI_READ, RUI_WRITE, SLI_BID, SLI_CLOSE, SLI_OPEN, SLI_RECEIVE

SLI_SEND_EX

The **SLI_SEND_EX** verb sends responses, SNA commands, and data from a Windows LUA application to a host LU.

The SLI_SEND_EX verb also supports inbound chaining. The maximum length of data that can be sent by a single verb is 4,294,967,295 bytes. This is compared to a maximum of 65,535 bytes that can be sent by the SLI_SEND verb.

The following structure describes the LUA_COMMON member of the VCB used by SLI_SEND_EX.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

The following union describes the **LUA_SPECIFIC** member of the VCB used by **SLI_SEND_EX**. Other union members are omitted for clarity.

```
union LUA_SPECIFIC {
    struct SLI_SEND_EX_SPECIFIC {
        unsigned char lua_sequence_number[2];
        unsigned long lua_data_length_ex;
    };
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_SEND_EX.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_SEND only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_SEND_EX** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_SEND_EX** and should be set to zero.

lua_data_length
This parameter is reserved and must be set to zero.

The length of data to be sent is set in the lua_data_length_ex parameter.

lua_data_ptr
Pointer to the application-supplied buffer that contains the data to be sent to the host by **SLI_SEND_EX**.

Both SNA commands and data are placed in this buffer, and they can be in an EBCDIC format.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are set for write functions and returned for read and bid functions. Its subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh
Supplied parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for RUI_WRITE and **SLI_SEND**, and returned by RUI_READ and RUI_BID. For the RH for **SLI_SEND_EX**, all fields except the queued-response indicator (**lua_rh.qri**) and pacing indicator (**lua_rh.pi**) are used.

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits.

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

Set one of the following flags to 1 to indicate on which message flow the data is to be sent:

lua_flag1.sscp_exp

lua_flag1.sscp_norm

lua_flag1.lu_exp

lua_flag1.lu_norm

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. This is a supplied parameter for **SLI_SEND_EX**.

Possible values are as follows:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_RSP

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SIGNAL

The SLI receives and responds to the BIND and STSN requests through the LUA interface extension routines.

LU-DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that RUI_BID was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Not used by **SLI_SEND_EX** and should be set to zero.

lua_sequence_number
The union member of **LUA_SPECIFIC** used by **SLI_SEND_EX**. Returned parameter. Contains the sequence number for either the first in the chain request unit or the only segment in the chain request unit. Note that this parameter is not byte-reversed.

lua_data_length_ex
The union member of **LUA_SPECIFIC** used by **SLI_SEND_EX**. Supplied parameter. Specifies the length of data being sent.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_PARAMETER_CHECK
Primary return code; the verb did not execute because of a parameter error.

LUA_INVALID_LUNAME

Secondary return code; an invalid **lua_luname** was specified.

LUA_BAD_SESSION_ID

Secondary return code; an invalid value for **lua_sid** was specified in the VCB.

LUA_BAD_DATA_PTR

Secondary return code; the **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

LUA_RESERVED_FIELD_NOT_ZERO

Secondary return code; a reserved parameter for the verb just issued is not set to zero.

LUA_INVALID_POST_HANDLE

Secondary return code; for a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

LUA_INVALID_FLOW

Secondary return code; the **lua_flag1** flow flags were set incorrectly when a verb was issued:

- When issuing **SLI_SEND_EX** to send an SNA response, set only one **lua_flag1** flow flag.
- When issuing SLI_RECEIVE_EX, set at least one lua_flag1 flow flag.

LUA_VERB_LENGTH_INVALID

Secondary return code; an LUA verb was issued with a value for **lua_verb_length** unexpected by LUA.

LUA_REQUIRED_FIELD_MISSING

Secondary return code; the verb that was issued either did not include a data pointer (if the data count was not zero) or did not include an **lua_flag1** flow flag.

LUA_INVALID_MESSAGE_TYPE

Secondary return code; the **lua_message_type** parameter is not recognized by the LUA interface.

LUA_DATA_LENGTH_ERROR

Secondary return code; the application did not provide user-supplied data required by the verb issued. Note that when **SLI_SEND_EX** is issued for an SNA LUSTAT command, status (in four bytes) is required, and that when SLI_OPEN is issued with secondary initialization, data is required.

LUA_STATE_CHECK
Primary return code; the verb did not execute because it was issued in an invalid state.

LUA_NO_SLI_SESSION

Secondary return code; a session was not open or was down due to an SLI_CLOSE or session failure when a command was issued.

LUA_MAX_NUMBER_OF_SENDS

Secondary return code; the application issued a third **SLI_SEND** or an **SLI_SEND_EX** before one completed.

LUA_SEND_ON_FLOW_PENDING

Secondary return code; an **SLI_SEND** or an **SLI_SEND_EX** was still outstanding when the application issued another **SLI_SEND_EX** for an SNA flow.

LUA_SESSION_FAILURE
Primary return code; an error condition, specified in the secondary return code, caused the session to fail.

LUA_RECEIVED_UNBIND

Secondary return code; the primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_NO_RUI_SESSION

Secondary return code; no session has been initialized for the LUA verb issued, or some verb other than SLI_OPEN was issued before the session was initialized.

LUA_LU_COMPONENT_DISCONNECTED

Secondary return code; an LU component is unavailable because it is not connected properly. Make sure that the power is on.

LUA_DATA_SEGMENT_LENGTH_ERROR

Secondary return code; one of the following has occurred:

- The supplied data segment for SLI_RECEIVE_EX or **SLI_SEND_EX** is not a read/write data segment as required.
- The supplied data segment for SLI_RECEIVE_EX is not as long as that provided in lua_max_length_ex.
- The supplied data segment for SLI_SEND_EX is not as long as that provided in lua_data_length_ex.

LUA_VERB_RECORD_SPANS_SEGMENTS

Secondary return code; the LUA VCB length parameter plus the segment offset is beyond the segment end.

LUA_NOT_ACTIVE

Secondary return code; LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

LUA_SLI_LOGIC_ERROR

Secondary return code; the LUA interface found an internal error in logic.

LUA_INVALID_PROCESS

Secondary return code; the session for which an LUA verb was issued is unavailable because another OS/2 process owns the session.

LUA_LU_INOPERATIVE

Secondary return code; a severe error occurred while the LUA was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

LUA_MODE_INCONSISTENCY

Secondary return code; performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

LUA_INSUFFICIENT_RESOURCES

Secondary return code; a temporary condition of insufficient resources caused the request receiver to be unable to perform. The request sent to the half-session component was not executed, even though it was understood and supported.

LUA_SEND_CORRELATION_TABLE_FULL

Secondary return code; the session send correlation table for the flow requested reached its capacity.

LUA_RU_LENGTH_ERROR

Secondary return code; the RU request was an incorrect length (either too short or too long). The request unit was not interpreted or processed even though it was delivered to the half-session component. The half-session capabilities do not match. This SNA sense code is also an exception request sense code.

LUA_FUNCTION_NOT_SUPPORTED

Secondary return code; LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

LUA_HDX_BRACKET_STATE_ERROR

Secondary return code; the existing state error prevented the current request from being sent. The determination was made by a protocol computer.

LUA_RESPONSE_ALREADY_SENT

Secondary return code; a response for the chain was already sent so that the current request was not sent. The determination was made by a protocol computer.

LUA_EXR_SENSE_INCORRECT

Secondary return code; the application responded negatively to an exception request. The sense code was unacceptable.

LUA_RESPONSE_OUT_OF_ORDER

Secondary return code; the current response was not for the oldest request. The determination was made by a protocol computer.

LUA_CHAIN_RESPONSE_REQUIRED

Secondary return code; a CHASE response was still outstanding when a more recent request was attempted. The determination was made by a protocol computer.

LUA_BRACKET

Secondary return code; the sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DIRECTION

Secondary return code; while the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_RESET

Secondary return code; a half-session of an active session but with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_QUIESCED

Secondary return code; a DFC or FMD request was received from a half-session that sent either a SHUTC command or QC command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_DATA_TRAFFIC_NOT_RESET

Secondary return code; while the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_NO_BEGIN_BRACKET

Secondary return code; the receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_SC_PROTOCOL_VIOLATION

Secondary return code; a violation of SC protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or data flow control state was found. Delivery to the half-session component was prevented.

LUA_IMMEDIATE_REQUEST_MODE_ERROR

Secondary return code; the request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_QUEUED_RESPONSE_ERROR

Secondary return code; the request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_ERP_SYNC_EVENT_ERROR

Secondary return code; a violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_BEFORE_SENDING_REQ

Secondary return code; a previously received request has not been responded to yet and an attempt was made in half-duplex send/receive mode to send a normal flow request. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

LUA_RSP_CORRELATION_ERROR

Secondary return code; a response was sent that does not correspond to a previously received request or a response was received that does not correspond to a previously sent request.

LUA_BB_NOT_ALLOWED

Secondary return code; the begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EB_NOT ALLOWED

Secondary return code; the end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_EXCEPTION_RSP_NOT_ALLOWED

Secondary return code; when an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_DEFINITE_RSP_NOT_ALLOWED

Secondary return code; when a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_ALLOWED

Secondary return code; the change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_RESPONSE_NOT_ALLOWED

Secondary return code; a request other than an EXR contained a "no response." The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CHAINING_NOT_SUPPORTED

Secondary return code; the chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_BRACKETS_NOT_SUPPORTED

Secondary return code; the bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_CD_NOT_SUPPORTED

Secondary return code; the change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_FI

Secondary return code; the format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_ALTERNATE_CODE_NOT_SUPPORTED

Secondary return code; the code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_REQUEST_CODE

Secondary return code; the request code was incorrectly specified. The BIND options chosen previously or the architectural rules

were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_SPEC_OF_SDI_RTI

Secondary return code; the SDI and the RTI were not specified correctly on a response. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_DR1I_DR2I_ERI

Secondary return code; the DR1I, the DR2I, and the ERI were specified incorrectly. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_QRI

Secondary return code; the queued response indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_EDI

Secondary return code; the EDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_INCORRECT_USE_OF_PDI

Secondary return code; the PDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

LUA_NO_SESSION

Secondary return code; a request to activate a session is required because no active half-session in the receiving end node for the origination-destination pair exists, or no active boundary function half-session component for the origination-destination pair in a node that supplies the boundary function exists. Delivery of the request could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, that usually indicates there is no longer a valid path to the session partner.

LUA_CANCELED
Primary return code; the secondary return code gives the reason for canceling the command.

LUA_TERMINATED

Secondary return code; the session was terminated when a verb was pending. The verb process has been canceled.

LUA_IN_PROGRESS
Primary return code; an asynchronous command was received but is not completed.

LUA_COMM_SUBSYSTEM_ABENDED
Primary return code; indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node has been broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

LUA_COMM_SUBSYSTEM_NOT_LOADED
Primary return code; a required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

LUA_INVALID_VERB_SEGMENT

Primary return code; the VCB extended beyond the end of the data segment.

LUA_UNEXPECTED_DOS_ERROR

Primary return code; after issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

LUA_STACK_TOO_SMALL

Primary return code; the stack size of the application is too small to execute the verb. Increase the stack size of your application.

LUA_INVALID_VERB

Primary return code; either the verb code or the operation code, or both, is invalid. The verb did not execute.

Remarks

SLI_SEND_EX sends responses, SNA commands, and data from the Windows LUA application to a host LU.

The difference between SLI_SEND_EX and SLI_SEND is that the SLI_SEND_EX verb supports inbound chaining and can send up to a maximum of 4,295 KB in a single verb request. In contrast, SLI_SEND is limited to sending up to 64 KB in a verb request. A single SLI_SEND_EX or SLI_SEND verb defines a chain. A single SLI_RECEIVE_EX or SLI_RECEIVE verb receives a whole chain.

A session must already be open to issue SLI_SEND_EX for a particular LU-LU session flow. To send data on the SSCP normal flow prior to the completion of SLI_OPEN, the session must have been initialized as primary with SSCP access. In addition, the session status must be INIT_COMPLETE.

The settings for lua_message_type determine the type of processing that will be done by SLI_SEND_EX. The following table indicates the parameters to set based on the value of lua_message_type.

SLI_SEND_EX parameter	LU_DATA SSCP_DATA	BID BIS RTR	CHASE QC	LUSTAT_LU LUSTAT_SSCP	QEC RELQ SBI SIGNAL	RQR	RSP
lua_data_length	Req.	0	0	Req.	0	0	Req. (0 if no data)
lua_data_ptr	Req. (0 if no data)	0	0	Req.	0	0	Req. (0 if no data)
lua_flag1 flow flags	0	0	0	0	0	0	Req. (set one)
lua_rh	FI DRL1 DRL2 RI BBI EBI CDI CSI EDI	SDI QRI	SDI QRI EBI CDI	SDI QRI DRL1 DRL2 RI BBI EBI CDI	SDI	0	RRI RI
lua_th	0	0	0	0	0	0	SNF

The location provided in **lua_data_ptr** and the length provided in **lua_data_length_ex** determine the data that the SLI sends. The data will be chained by the SLI verbs if necessary.

When sending a response, the type of response determines the SLI_SEND_EX information required. For all responses, you must:

- Set the selected **lua_flag1** flow flag.
- Provide the sequence number in lua_th.snf for the request to which you are responding.
- Set lua_message_type to LUA_MESSAGE_TYPE_RSP.

For multichain message responses, the sequence number of the last received chain element must be used. For a response to a multichain message ending with a CANCEL command, the CANCEL command sequence number is used.

For positive responses that only require the request code, set lua_rh.ri to zero (indicating that the response is positive) and lua_data_length to zero (indicating no data is provided). The request code is filled in by the SLI, using the sequence number provided.

For negative responses in which lua_rh.ri is set to 1, set the lua_data_ptr to the SNA sense code address and the lua_data_length to the SNA sense code length (four bytes). The sequence number is used by the SLI to fill in the request code.

See Also

RUI_INIT, RUI_READ, RUI_WRITE, SLI_BID, SLI_CLOSE, SLI_OPEN, SLI_RECEIVE_EX

SLI_BIND_ROUTINE

The **SLI_BIND_ROUTINE** verb notifies the Windows LUA application that a BIND request has come from the host and allows the user-supplied routine to examine the request and formulate a response.

The following structure describes the LUA_COMMON member of the VCB used by SLI_BIND_ROUTINE.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_BIND_ROUTINE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_BIND_ROUTINE only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_BIND_ROUTINE** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_BIND_ROUTINE** and should be set to zero.

lua_data_length
Returned parameter. Specifies the length of the BIND RU data returned in the data buffer.

lua_data_ptr
For the **SLI_BIND_ROUTINE** this parameter contains the address of the BIND RU.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Supplied parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are returned for read and bid functions.

lua_rh
Supplied parameter. Contains the SNA request/response header (RH) of the message sent or received.

lua_flag1
Supplied parameter. Contains a data structure containing flags for messages supplied by the application.

lua_message_type
Supplied parameter. Specifies the type of SNA data or command sent to the host.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Not used by **SLI_BIND_ROUTINE** and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_NEGATIVE_RSP
Primary return code; either the LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

Remarks

SLI_BIND_ROUTINE provides a mechanism for the Windows LUA application to examine BIND requests that are received from the host. The Windows LUA uses a user-supplied DLL to notify the Windows LUA application that a BIND request has been received. The user-supplied DLL routine then examines the contents of the BIND and formulates a response for the request.

The DLL name for the routine is provided as extensions of the SLI_OPEN verb's VCB. The lua_extension_list_offset parameter provides the offset from the start of the VCB to the first name in the extension list.

The Windows LUA interface assigns storage space where the VCB is structured. The VCB of SLI_BIND_ROUTINE contains lua_th

and lua_rh. The address of the BIND RU is specified in lua_data_ptr and the length of the RU is specified in lua_data_length.

When SLI_BIND_ROUTINE returns to the Windows LUA, processing of SLI_BIND_ROUTINE is completed. The BIND response should overwrite the BIND RU. When the BIND is accepted, the primary return code should be set to LUA_OK. If the BIND is rejected, the primary return code should be set to LUA_NEGATIVE_RSP and the BIND buffer contains the negative sense code. The lua_data_ptr parameter should not be modified.

If a negative response is returned from SLI_BIND_ROUTINE, SLI_OPEN is canceled. The lua_prim_rc of the SLI_OPEN is set to LUA_SESSION_FAILURE, and the lua_sec_rc is set to LUA_NEG_RSP_FROM_BIND_ROUTINE.

See Also

RUI_INIT, RUI_PURGE, RUI_READ, RUI_WRITE, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

SLI_STSN_ROUTINE

The **SLI_STSN_ROUTINE** verb notifies the Windows LUA application that an STSN command has come from the host and allows the user-supplied routine to examine the request and formulate a response.

The following structure describes the LUA_COMMON member of the VCB used by SLI_STSN_ROUTINE.

```
struct LUA_COMMON {
    unsigned short    lua_verb;
    unsigned short    lua_verb_length;
    unsigned short    lua_prim_rc;
    unsigned long     lua_sec_rc;
    unsigned short    lua_opcode;
    unsigned long     lua_correlator;
    unsigned char     lua_luname[8];
    unsigned short    lua_extension_list_offset;
    unsigned short    lua_cobol_offset;
    unsigned long     lua_sid;
    unsigned short    lua_max_length;
    unsigned short    lua_data_length;
    char FAR *        lua_data_ptr;
    unsigned long     lua_post_handle;
    struct LUA_TH     lua_th;
    struct LUA_RH     lua_rh;
    struct LUA_FLAG1  lua_flag1;
    unsigned char     lua_message_type;
    struct LUA_FLAG2  lua_flag2;
    unsigned char     lua_resv56[7];
    unsigned char     lua_encr_decr_option;
};
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_SLI for SLI verbs.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, LUA_OPCODE_SLI_STSN_ROUTINE.

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_STSN_ROUTINE only requires this parameter if lua_sid is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Not used by **SLI_STSN_ROUTINE** and should be set to zero.

lua_cobol_offset
Not used by LUA in Microsoft® Host Integration Server or Microsoft® SNA Server and should be zero.

lua_sid
Supplied parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Not used by **SLI_STSN_ROUTINE** and should be set to zero.

lua_data_length
Returned parameter. Specifies the length of the STSN RU data returned in the data buffer.

lua_data_ptr
For the **SLI_STSN_ROUTINE** this parameter contains the address of the STSN RU.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message received. Various subparameters are returned for read and bid functions.

lua_rh
Returned parameter. Contains the SNA request/response header (RH) of the message sent or received.

lua_flag1
Supplied parameter. Contains a data structure containing flags for messages supplied by the application.

lua_message_type
Supplied parameter. Specifies the type of SNA data or command sent to the host.

lua_flag2
Returned parameter. Contains flags for messages returned by LUA.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56
Reserved and should be set to zero.

lua_encr_decr_option
Not used by **SLI_STSN_ROUTINE** and should be set to zero.

Return Codes

LUA_OK
Primary return code; the verb executed successfully.

LUA_SEC_OK

Secondary return code; no additional information exists for LUA_OK.

LUA_NEGATIVE_RSP
Primary return code; either the LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

Remarks

SLI_STSN_ROUTINE provides a mechanism for the Windows LUA application to examine and respond to STSN commands. The Windows LUA notifies the Windows LUA application that an STSN command has been received from the host. This is done through a user-supplied DLL. The user's DLL examines the STSN request and formulates a response to the request.

The DLL name for the routine is provided as extensions of the SLI_OPEN verb's VCB. The lua_extension_list_offset parameter provides the offset from the start of the VCB to the first name in the extension list.

The Windows LUA interface assigns storage space where the VCB is structured. The VCB of the SLI_STSN_ROUTINE contains

lua_th and lua_rh. The address of the STSN RU is specified in lua_data_ptr and the length of the RU is specified in lua_data_length.

When SLI_STSN_ROUTINE returns to the Windows LUA, processing of the SLI_STSN_ROUTINE is completed. The STSN response should overwrite the STSN RU. When the STSN is accepted, the primary return code should be set to LUA_OK. If the STSN is rejected, the primary return code should be set to LUA_NEGATIVE_RSP and the STSN buffer contains the negative sense code. The lua_data_ptr parameter should not be modified.

If a negative response is returned from SLI_STSN_ROUTINE, SLI_OPEN is canceled. The lua_prim_rc of the SLI_OPEN is set to LUA_SESSION_FAILURE, and the lua_sec_rc is set to LUA_NEG_RSP_FROM_STSN_ROUTINE.

See Also

RUI_INIT, RUI_PURGE, RUI_READ, RUI_WRITE, SLI_OPEN, SLI_PURGE, SLI_RECEIVE, SLI_SEND

LUA Extensions for the Windows Environment

The extensions described in this section are designed for all implementations and versions of the Microsoft® Windows® graphical environment version 3.0 or later. They provide support for maximum Windows-based programming compatibility and optimum application performance in both 16-bit and 32-bit operating environments.

Windows LUA allows multithreaded Windows-based processes. A process contains one or more threads of execution. The 16-bit Windows environment is not multithreaded. In this instance, a task corresponds to a process with a single thread. All references to threads in this document refer to actual threads in multithreaded Windows environments.

For each extension, this section provides a definition of the function with syntax, return codes, and remarks for using the extension.

These functions can be grouped into two categories depending on whether RUI or SLI verbs are used.

Functions for use with RUI verbs are:

RUI
: Provides event notification for all RUI verbs.

WinRUI
: Provides asynchronous message notification for all Windows-based RUI verbs.

WinRUICleanup
: Terminates and deregisters an application using RUI verbs from a Windows LUA implementation.

WinRUIGetLastInitStatus
: Enables an application using RUI verbs to initiate status reporting, terminate status reporting, or find the current status.

WinRUIStartup
: Allows an application using RUI verbs to specify the version of Windows LUA required and to retrieve details of the specific Windows LUA implementation. This function must be called by an application to register itself with a Windows LUA implementation before issuing any further Windows LUA calls.

Functions for use with SLI verbs are:

SLI
: Provides event notification for all SLI verbs.

WinSLI
: Provides asynchronous message notification for all Windows-based SLI verbs.

WinSLICleanup
: Terminates and deregisters an application using SLI verbs from a Windows LUA implementation.

WinSLIStartup
: Allows an application using SLI verbs to specify the version of Windows LUA required and to retrieve details of the specific Windows LUA implementation. This function must be called by an application to register itself with a Windows LUA implementation before issuing any further Windows LUA calls.

RUI

The **RUI** function provides event notification for all RUI verbs.

```
void WINAPI RUI(
  LUA_VERB_RECORD FAR *lpVCB
);
```

Parameters

lpVCB
Pointer to the LUA VCB, **LUA_VERB_RECORD**.

Return Values

The code returned in **lua_prim_rc** indicates whether asynchronous notification will occur. If the field is set to LUA_IN_PROGRESS, asynchronous notification will occur through event signaling. If the flag is not LUA_IN_PROGRESS, the request completed synchronously. Examine the primary return code and secondary return code for any errors.

Remarks

The application must provide a handle to an event in the **lua_post_handle** parameter of the VCB. The event must be in the not-signaled state.

When the asynchronous operation is complete, the application is notified through the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions.

See Also

WinRUI

SLI

The **SLI** function provides event notification for all SLI verbs.

```
void WINAPI SLI(
  LUA_VERB_RECORD FAR *lpVCB
);
```

Parameters

lpVCB
Pointer to the LUA VCB, **LUA_VERB_RECORD**.

Return Values

The code returned in **lua_prim_rc** indicates whether asynchronous notification will occur. If the field is set to LUA_IN_PROGRESS, asynchronous notification will occur through event signaling. If the flag is not LUA_IN_PROGRESS, the request completed synchronously. Examine the primary return code and secondary return code for any errors.

Remarks

The application must provide a handle to an event in the **lua_post_handle** parameter of the VCB. The event must be in the not-signaled state.

When the asynchronous operation is complete, the application is notified through the signaling of the event. Upon signaling of the event, examine the primary return code and secondary return code for any error conditions.

See Also

WinSLI

WinRUI

The **WinRUI** function provides asynchronous message notification for all Windows-based RUI verbs.

```
int WINAPI WinRUI(
  HWND hWnd,
  LUA_VERB_RECORD FAR *lpVCB
);
```

Parameters

hWnd
Handle of window to receive message.
lpVCB
Pointer to the LUA VCB, **LUA_VERB_RECORD**.

Return Values

The function returns a value indicating whether the request was accepted by the Windows-based RUI for processing. A returned value of zero indicates that the request was accepted and will be processed. A value other than zero indicates an error. Possible error codes are as follows:

WLUAINVALIDHANDLE
The window handle provided is invalid.
WLUASTARTUPNOTCALLED
The application has not initiated a session using WinRUIStartup.

The value returned in **lua_flag2.async** indicates whether asynchronous notification will occur. If the flag is set (nonzero), asynchronous notification will occur through a message posted to the application's message queue. If the flag is not set, the request completed synchronously. Examine the primary return code and secondary return code for any error conditions.

Remarks

When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with "WinRUI" as the input string. The *lParam* argument contains the address of the VCB being posted as complete. The *wParam* argument is undefined.

> **Note** It is possible for the request to be accepted for processing (the function call returns zero) but rejected later with a primary return code and secondary return code set in the VCB. Examine the primary return code and secondary return code for any error conditions.

If the application calls **WinRUI** without first initializing the session using **WinRUIStartup**, an error is returned.

See Also

RUI, WinRUIStartup

WinRUICleanup

The **WinRUICleanup** function terminates and deregisters an application using RUI verbs from a Windows LUA implementation.

```
BOOL WINAPI WinRUICleanup(void);
```

Return Values

The return code specifies whether the deregistration was successful. If the value is not zero, the application was successfully deregistered. If the value is zero, the application was not deregistered.

Remarks

Use **WinRUICleanup** to indicate deregistration of a Windows LUA application from a Windows LUA implementation. This function can be used, for example, to free up resources allocated to the specific application.

If **WinRUICleanup** is called while LUs are in session (RUI_TERM not issued), the cleanup code should issue an **RUI_TERM** close type ABEND for the application for all open sessions.

See Also

RUI_TERM, WinRUIStartup

WinRUIGetLastInitStatus

The **WinRUIGetLastInitStatus** function enables an application to determine the status of an RUI_INIT, so that the application can evaluate whether the **RUI_INIT** should be timed out. This extension can be used to initiate status reporting, terminate status reporting, or find the current status. For details, see the Remarks section.

```
int WINAPI WinRUIGetLastInitStatus(
  DWORD dwSid,
  HANDLE hStatusHandle,
  DWORD dwNotifyType,
  BOOL bClearPrevious
);
```

Parameters

dwSid
Specifies the RUI session identifier of the session for which status will be determined. If *dwSid* is zero, *hStatusHandle* is used to report status on all sessions. Note that the **lua_sid** parameter in the RUI_INIT VCB is valid as soon as the call to RUI or WinRUI for the **RUI_INIT** returns.

hStatusHandle
Specifies a handle used for signaling the application that the status for the session (specified by *dwSid*) has changed. Can be a window handle, an event handle, or NULL; *dwNotifyType* must be set accordingly:

- If *hStatusHandle* is a window handle, status is sent to the application through a window message. The message is obtained from **RegisterWindowMessage** using the string "WinRUI". The parameter *wParam* contains the session status (see Return Codes). Depending on the value of *dwNotifyType*, *lParam* contains either the RUI session identifier of the session, or the value of **lua_correlator** from the RUI_INIT verb.
- If *hStatusHandle* is an event handle, when the status for the session specified by *dwSid* changes, the event is put into the signaled state. The application must then make a further call to **WinRUIGetLastInitStatus** to find out the new status. Note that the event should not be the same as one used for signaling completion of any RUI verb.
- If *hStatusHandle* is NULL, the status of the session specified by *dwSid* is returned in the return code. In this case, *dwSid* must not be zero unless *bClearPrevious* is TRUE. If *hStatusHandle* is NULL, *dwNotifyType* is ignored.

dwNotifyType
Specifies the type of indication required. This determines the contents of the *lParam* of the window message, and how **WinRUIGetLastInitStatus** interprets *hStatusHandle*. Allowed values are:

WLUA_NTFY_EVENT

The *hStatusHandle* parameter contains an event handle.

WLUA_NTFY_MSG_CORRELATOR

The *hStatusHandle* parameter contains a window handle, and the *lParam* of the returned window message should contain the value of the **lua_correlator** field on the RUI_INIT.

WLUA_NTFY_MSG_SID

The *hStatusHandle* parameter contains a window handle, and the *lParam* of the returned window message should contain the LUA session identifier.

bClearPrevious
If TRUE, status messages are no longer sent for the session identified by *dwSid*. If *dwSid* is zero, status messages are no longer sent for any session. If *bClearPrevious* is TRUE, *hStatusHandle* and *dwNotifyType* are ignored.

Return Values

WLUASYSNOTREADY
SNABASE is not running.
WLUANTFYINVALID
The *dwNotifyType* parameter is invalid.
WLUAINVALIDHANDLE
The *hStatusHandle* parameter does not contain a valid handle.
WLUASTARTUPNOTCALLED
WinRUIStartup has not been called.

WLUALINKINACTIVE
The link to the host is not yet active.
WLUALINKACTIVATING
The link to the host is being activated.
WLUAPUINACTIVE
The link to the host is active, but no ACTPU has yet been received.
WLUAPUACTIVE
An ACTPU has been received.
WLUAPUREACTIVATED
The PU has been reactivated.
WLUALUINACTIVE
The link to the host is active, and an ACTPU has been received, but no ACTLU has been received.
WLUALUACTIVE
The LU is active.
WLUALUREACTIVATED
The LU has been reactivated.
WLUAUNKNOWN
The session is in an unknown status. (This is an internal error.)
WLUAGETLU
The session is waiting for an Open(SSCP) response from the node.
WLUASIDINVALID
The SID specified does not match any known by the RUI.
WLUASIDZERO
The *hStatusHandle* parameter is NULL and *bClearPrevious* is FALSE, but *dwSid* is zero.
WLUAGLOBALHANDLER
The *dwSid* parameter is zero, and messages from all sessions will be notified. (This is a normal return code, not an error.)

Remarks

This extension is intended to be used with either a window handle or an event handle to enable asynchronous notification of status changes. It can also be used alone to find out the current status of a session.

With a window handle

There are two ways to use this extension with a window handle:

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NTFY_MSG_CORRELATOR,FALSE);
```

—or—

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NTFY_MSG_SID,FALSE);
```

With this implementation, changes in status are reported by a window message sent to the window handle specified. If WLUA_NTFY_MSG_CORRELATOR is specified, the *lParam* field in the window message contains the **lua_correlator** field for the session. If WLUA_NTFY_MSG_SID is specified, the *lParam* field in the window message contains the LUA session identifier for the session.

When the extension has been used with a window handle, use the following to cancel status reporting:

```
WinRUIGetLastInitStatus(Sid,NULL,0,TRUE);
```

For this implementation, note that if *Sid* is nonzero, status is only reported for that session. If *Sid* is zero, status is reported for all sessions.

With an event handle

To use this extension with an event handle, implement it as follows:

```
WinRUIGetLastInitStatus(Sid,Handle,WLUA_NOTIFY_EVENT,FALSE);
```

The event whose handle is given will be signaled when a change in state occurs. Since no information is returned when an event is signaled, a further call must be issued to find out the status.

```
Status = WinRUIGetLastInitStatus(Sid,NULL,0,0,FALSE);
```

Note that in this case, a *Sid* must be specified.

When the extension has been used with an event handle, use the following to cancel the reporting of status:

```
WinRUIGetLastInitStatus(Sid,NULL,0,TRUE);
```

To query current status

To use this extension to query the current status of a session, it is not necessary to use an event or window handle. Instead, use the following:

```
Status = WinRUIGetLastInitStatus(Sid,NULL,0,0,FALSE);
```

See Also

RUI, RUI_INIT, WinRUI, WinRUIStartup

WinRUIStartup

The **WinRUIStartup** function allows an application using RUI verbs to specify the version of Windows LUA required and to retrieve details of the specific Windows LUA implementation. This function must be called by an application to register itself with a Windows LUA implementation before issuing any further Windows LUA calls.

```
int WINAPI WinRUIStartup(
  WORD wVersionRequired,
  LUADATA FAR *lpLuaData
);
```

Parameters

wVersionRequired
Specifies the version of Windows LUA support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

lpLuaData
Pointer to the **LUADATA** structure containing the returned version number information.

Return Values

The return code specifies whether the application was registered successfully and whether the Windows LUA implementation can support the specified version number. If the value is zero, it was registered successfully and the specified version can be supported. Otherwise, the return code is one of the following:

WLUASYSNOTREADY
The underlying network subsystem is not ready for network communication.

WLUAVERNOTSUPPORTED
The version of Windows LUA support requested is not provided by this particular Windows LUA implementation.

WLUAINVALID
The Windows LUA version specified by the application is not supported by this DLL.

WLUAFAILURE
A failure occurred while the Windows LUA DLL was initializing. This usually occurs because an operating system call failed.

WLUAINITREJECT
WinRUIStartup was called multiple times.

Remarks

To support future Windows LUA implementations and applications that may have functionality differences, a negotiation takes place in **WinRUIStartup**. An application passes to **WinRUIStartup** the Windows LUA version that it can use. If this version is lower than the lowest version supported by the Windows LUA DLL, the DLL cannot support the application and **WinRUIStartup** fails. If the version is not lower, however, the call succeeds and returns the highest version of Windows LUA supported by the DLL. If this version is lower than the lowest version supported by the application, the application either fails its initialization or attempts to find another Windows LUA DLL on the system.

This negotiation allows both a Windows LUA DLL and a Windows LUA application to support a range of Windows LUA versions. An application can successfully use a DLL if there is any overlap in the versions. The following table illustrates how **WinRUIStartup** works in conjunction with different application and DLL versions:

App versions	DLL versions	To WinRUIStartup	From WinRUIStartup	Result
1.0	1.0	1.0	1.0	Use 1.0
1.0, 2.0	1.0	2.0	1.0	Use 1.0
1.0	1.0, 2.0	1.0	2.0	Use 1.0
1.0	2.0, 3.0	1.0	WLUAINVALID	Fail
2.0, 3.0	1.0	3.0	1.0	App fails
1.0, 2.0, 3.0	1.0, 2.0, 3.0	3.0	3.0	Use 3.0

Note The application that uses RUI verbs must call **WinRUIStartup** prior to issuing any other LUA commands. However, **WinRUIStartup** needs to be called only once per application. If it is called multiple times, the subsequent calls will be rejected.

Details of the actual LUA implementation are described in the **WLUADATA** structure, defined as follows:

```
typedef struct { WORD wVersion;
            char szDescription[WLUADESCRIPTION_LEN+1];
            } LUADATA;
```

Having made its last Windows LUA call, an application should call the **WinRUICleanup** routine.

Each LUA application that uses RUI verbs must make a **WinRUIStartup** call before issuing any other LUA calls.

See Also

WinRUICleanup

WinSLI

The **WinSLI** function provides asynchronous message notification for all Windows-based SLI verbs.

```
int WINAPI WinSLI(
  HWND hWnd,
  LUA_VERB_RECORD FAR *lpVCB
);
```

Parameters

hWnd
Handle of window to receive message.
lpVCB
Pointer to the LUA VCB, **LUA_VERB_RECORD**.

Return Values

The function returns a value indicating whether the request was accepted by the Windows-based SLI for processing. A returned value of zero indicates that the request was accepted and will be processed. A value other than zero indicates an error. Possible error codes are as follows:

WLUAINVALIDHANDLE
The window handle provided is invalid.
WLUASTARTUPNOTCALLED
The application has not initiated a session using WinSLIStartup.

The value returned in **lua_flag2.async** indicates whether asynchronous notification will occur. If the flag is set (nonzero), asynchronous notification will occur through a message posted to the application's message queue. If the flag is not set, the request completed synchronously. Examine the primary return code and secondary return code for any error conditions.

Remarks

When the asynchronous operation is complete, the application's window *hWnd* receives the message returned by **RegisterWindowMessage** with 'WinSLI" as the input string. The *lParam* argument contains the address of the VCB being posted as complete. The *wParam* argument is undefined.

> **Note** It is possible for the request to be accepted for processing (the function call returns zero) but rejected later with a primary return code and secondary return code set in the VCB. Examine the primary return code and secondary return code for any error conditions.

If the application calls **WinSLI** without first initializing the session using **WinSLIStartup**, an error is returned.

See Also

SLI, WinSLIStartup

WinSLICleanup

The **WinSLICleanup** function terminates and deregisters an application using SLI verbs from a Windows LUA implementation.

```
BOOL WINAPI WinSLICleanup(void);
```

Return Values

The return code specifies whether the deregistration was successful. If the value is not zero, the application was successfully deregistered. If the value is zero, the application was not deregistered.

Remarks

Use **WinSLICleanup** to indicate deregistration of a Windows LUA application from a Windows LUA implementation. This function can be used, for example, to free up resources allocated to the specific application.

If **WinSLICleanup** is called while LUs are in session (SLI_CLOSE not issued), the cleanup code should issue an **SLI_CLOSE** close type ABEND for the application for all open sessions.

See Also

SLI_CLOSE, WinSLIStartup

WinSLIStartup

The **WinSLIStartup** function allows an application using the SLI verbs to specify the version of Windows LUA required and to retrieve details of the specific Windows LUA implementation. This function must be called by an application to register itself with a Windows LUA implementation before issuing any further Windows LUA calls.

```
int WINAPI WinSLIStartup(
  WORD wVersionRequired,
  LUADATA FAR *lpLuaData
);
```

Parameters

wVersionRequired
Specifies the version of Windows LUA support required. The high-order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.

lpLuaData
Pointer to the **LUADATA** structure containing the returned version number information.

Return Values

The return code specifies whether the application was registered successfully and whether the Windows LUA implementation can support the specified version number. If the value is zero, it was registered successfully and the specified version can be supported. Otherwise, the return code is one of the following:

WLUASYSNOTREADY
The underlying network subsystem is not ready for network communication.

WLUAVERNOTSUPPORTED
The version of Windows LUA support requested is not provided by this particular Windows LUA implementation.

WLUAINVALID
The Windows LUA version specified by the application is not supported by this DLL.

WLUAFAILURE
A failure occurred while the Windows LUA DLL was initializing. This usually occurs because an operating system call failed.

WLUAINITREJECT
WinSLIStartup was called multiple times.

Remarks

To support future Windows LUA implementations and applications that may have functionality differences, a negotiation takes place in **WinSLIStartup**. An application passes to **WinSLIStartup** the Windows LUA version that it can use. If this version is lower than the lowest version supported by the Windows LUA DLL, the DLL cannot support the application and **WinSLIStartup** fails. If the version is not lower, however, the call succeeds and returns the highest version of Windows LUA supported by the DLL. If this version is lower than the lowest version supported by the application, the application either fails its initialization or attempts to find another Windows LUA DLL on the system.

This negotiation allows both a Windows LUA DLL and a Windows LUA application to support a range of Windows LUA versions. An application can successfully use a DLL if there is any overlap in the versions. The following table illustrates how **WinSLIStartup** works in conjunction with different application and DLL versions:

App versions	DLL versions	To WinSLIStartup	From WinSLIStartup	Result
1.0	1.0	1.0	1.0	Use 1.0
1.0, 2.0	1.0	2.0	1.0	Use 1.0
1.0	1.0, 2.0	1.0	2.0	Use 1.0
1.0	2.0, 3.0	1.0	WLUAINVALID	Fail
2.0, 3.0	1.0	3.0	1.0	App fails
1.0, 2.0, 3.0	1.0, 2.0, 3.0	3.0	3.0	Use 3.0

Note The application that uses SLI verbs must call **WinSLIStartup** prior to issuing any other LUA commands. However, **WinSLIStartup** needs to be called only once per application. If it is called multiple times, the subsequent calls will be rejected.

Details of the actual LUA implementation are described in the **WLUADATA** structure, defined as follows:

```
typedef struct { WORD wVersion;
            char szDescription[WLUADESCRIPTION_LEN+1];
            } LUADATA;
```

Having made its last Windows LUA call, an application should call the **WinSLICleanup** routine.

Each LUA application that uses SLI verbs must make a **WinSLIStartup** call before issuing any other LUA calls.

See Also

WinSLICleanup

SNA Server Enhancement to the Windows LUA Environment

This section describes the Microsoft® Host Integration Server and Microsoft® SNA Server-specific extension to Windows LUA that converts primary and secondary return codes in the verb control block (VCB) to a printable string.

Following is a definition of the function, syntax, returns, and remarks for using the extension.

GetLuaReturnCode

The **GetLuaReturnCode** function converts the primary and secondary return codes in the VCB to a printable string. This function provides a standard set of error strings for use by LUA applications.

```
int WINAPI GetLuaReturnCode(
    struct LUA_COMMON FAR *vpb,
    UINT buffer_length,
    unsigned char FAR *buffer_addr
);
```

Parameters

vpb
Supplied parameter. Specifies the address of the verb control block.

buffer_length
Supplied parameter. Specifies the length of the buffer pointed to by *buffer_addr*. The recommended length is 256.

buffer_addr
Supplied/returned parameter. Specifies the address of the buffer that will hold the formatted, null-terminated string.

Return Codes

0x20000001
The parameters are invalid; the function could not read from the specified verb control block or could not write to the specified buffer.

0x20000002
The specified buffer is too small.

0x20000003
The LUA string library LUASTR.DLL (for Windows) or LUAST32.DLL (for Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95) could not be loaded.

Remarks

The descriptive error string returned in *buffer_addr* does not terminate with a newline character (**\n**).

The descriptive error strings are contained in LUASTR.DLL (for Windows) or LUAST32.DLL (for Windows 2000, Windows NT, Windows 98, or Windows 95) and can be customized for different languages.

LUA Verb Control Blocks

When an application issues a Windows LUA verb, the verb is coded within the application as a precisely defined VCB. The total length of this VCB is variable and is defined by **lua_verb_length**.

This section defines the structure of individual Windows LUA VCBs.

This section contains:

Common Structure of LUA VCBs

The following data structure shows the parameters that are common to all Windows LUA verbs.

```
struct LUA_COMMON {
    unsigned short lua_verb;
    unsigned short lua_verb_length;
    unsigned short lua_prim_rc;
    unsigned long  lua_sec_rc;
    unsigned short lua_opcode;
    unsigned long  lua_correlator;
    unsigned char  lua_luname[8];
    unsigned short lua_extension_list_offset;
    unsigned short lua_cobol_offset;
    unsigned long  lua_sid;
    unsigned short lua_max_length;
    unsigned short lua_data_length;
    char FAR *     lua_data_ptr;
    unsigned long  lua_post_handle;
    struct LUA_TH  lua_th;
    struct LUA_RH  lua_rh;
    struct LUA_FLAG1 lua_flag1;
    unsigned char  lua_message_type;
    struct LUA_FLAG2 lua_flag2;
    unsigned char  lua_resv56[7];
    unsigned char  lua_encr_decr_option;
} LUA_COMMON;
```

Members

lua_verb
Supplied parameter. Contains the verb code, LUA_VERB_RUI for RUI verbs or LUA_VERB_SLI for SLI verbs. For both of these macros the value is 0x5200.

lua_verb_length
Supplied parameter. Specifies the length in bytes of the LUA VCB. It must contain the length of the verb record being issued.

lua_prim_rc
Primary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_sec_rc
Secondary return code set by LUA at the completion of the verb. The valid return codes vary depending on the LUA verb issued.

lua_opcode
Supplied parameter. Contains the LUA command code (verb operation code) for the verb to be issued, for example, LUA_OPCODE_RUI_BID for the RUI_BID verb. Valid values are as follows:

LUA_OPCODE_SLI_OPEN

LUA_OPCODE_SLI_CLOSE

LUA_OPCODE_SLI_RECEIVE

LUA_OPCODE_SLI_SEND

LUA_OPCODE_SLI_PURGE

LUA_OPCODE_SLI_BID

LUA_OPCODE_SLI_BIND_ROUTINE

LUA_OPCODE_SLI_STSN_ROUTINE

LUA_OPCODE_SLI_CRV_ROUTINE

LUA_OPCODE_RUI_INIT

LUA_OPCODE_RUI_TERM

LUA_OPCODE_RUI_READ

LUA_OPCODE_RUI_WRITE

LUA_OPCODE_RUI_PURGE

LUA_OPCODE_RUI_BID

lua_correlator
Supplied parameter. Contains a user-supplied value that links the verb with other user-supplied information. LUA does not use or change this information. This parameter is optional.

lua_luname
Supplied parameter. Specifies the ASCII name of the local LU used by the Windows LUA session.

SLI_OPEN and RUI_INIT require this parameter. Other Windows LUA verbs only require this parameter if **lua_sid** is zero.

This parameter is eight bytes long, padded on the right with spaces (0x20) if the name is shorter than eight characters.

lua_extension_list_offset
Specifies the offset from the start of the VCB to the extension list of user-supplied dynamic-link libraries (DLLs). This parameter is not used by RUI in Microsoft® Host Integration Server or Microsoft® SNA Server and should be set to zero. The value must be the beginning of a word boundary unless there is no extension list.

lua_cobol_offset
Offset of the Cobol extension. Not used by LUA in Host Integration Server or SNA Server and should be zero.

lua_sid
Supplied and returned parameter. Specifies the session identifier and is returned by SLI_OPEN and RUI_INIT. Other verbs use this parameter to identify the session used for the command. If other verbs use the **lua_luname** parameter to identify sessions, set the **lua_sid** parameter to zero.

lua_max_length
Specifies the length of received buffer for RUI_READ and SLI_RECEIVE. For other RUI and SLI verbs, it is not used and should be set to zero.

lua_data_length
Specifies the length of the data being sent or received. It specifies the length of data returned in **lua_peek_data** for the RUI_BID verb.

lua_data_ptr
Pointer to an application-supplied buffer.

When SLI_RECEIVE or RUI_READ is issued, this parameter points to the location to receive the data from the host.

When SLI_SEND or RUI_WRITE is issued, this parameter points to the location of the application's data to be sent to the host.

When SLI_PURGE or RUI_PURGE is issued, this parameter points to the location of the **SLI_RECEIVE** or **RUI_READ** verb's VCB that is to be canceled.

When SLI_OPEN is issued, this parameter can be one of the following:

- The logon message for the SSCP normal flow when the initialization type is secondary with an unformatted logon message.
- The RU for INITSELF. When the initialization type is secondary with INITSELF, the necessary data for the application is provided.
- For all other open types, this field should be set to zero.

For other RUI and SLI verbs, this parameter is not used and should be set to zero. Both SNA commands and data are placed in this buffer and they can be in an EBCDIC format.

This information is provided by the Windows LUA application.

lua_post_handle
Supplied parameter. Used under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95 if asynchronous notification is to be accomplished by events. This variable contains the handle of the event to be signaled or a window handle.

For all other environments, this parameter is reserved and should be set to zero.

lua_th
Returned parameter. Contains the SNA transmission header (TH) of the message sent or received. Various subparameters are set for write functions and returned for read and bid functions. The subparameters are as follows:

lua_th.flags_fid

Format identification type 2, four bits.

lua_th.flags_mpf

Segmenting mapping field, two bits. Defines the type of data segment. The following values are valid:

0x00 Middle segment
0x04 Last segment
0x08 First segment
0x0C Only segment

lua_th.flags_odai

Originating address field–destination address field (OAF–DAF) assignor indicator, one bit.

lua_th.flags_efi

Expedited flow indicator, one bit.

lua_th.daf

Destination address field (DAF), an unsigned char.

lua_th.oaf

Originating address field (OAF), an unsigned char.

lua_th.snf

Sequence number field, an unsigned char[2].

lua_rh

Returned parameter. Contains the SNA request/response header (RH) of the message sent or received. It is set for the write function and returned by the read and bid functions. Its subparameters are as follows:

lua_rh.rri

Request-response indicator, one bit.

lua_rh.ruc

RU category, two bits. The following values are valid:

LUA_RH_FMD (**0x00**) FM data segment
LUA_RH_NC (**0x20**) Network control
LUA_RH_DFC (**0x40**) Data flow control
LUA_RH_SC (**0x60**) Session control

lua_rh.fi

Format indicator, one bit.

lua_rh.sdi

Sense data included indicator, one bit.

lua_rh.bci

Begin chain indicator, one bit.

lua_rh.eci

End chain indicator, one bit.

lua_rh.dr1i

Definite response 1 indicator, one bit.

lua_rh.dr2i

Definite response 2 indicator, one bit.

lua_rh.ri

Exception response indicator (for a request), or response type indicator (for a response), one bit.

lua_rh.qri

Queued response indicator, one bit.

lua_rh.pi

Pacing indicator, one bit.

lua_rh.bbi

Begin bracket indicator, one bit.

lua_rh.ebi

End bracket indicator, one bit.

lua_rh.cdi

Change direction indicator, one bit.

lua_rh.csi

Code selection indicator, one bit.

lua_rh.edi

Enciphered data indicator, one bit.

lua_rh.pdi

Padded data indicator, one bit.

lua_flag1

Supplied parameter. Contains a data structure containing flags for messages supplied by the application. This parameter is used by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, SLI_RECEIVE, and SLI_SEND. For other LUA verbs this parameter is not used and should be set to zero. Its subparameters are as follows:

lua_flag1.bid_enable

Bid enable indicator, one bit.

lua_flag1.close_abend

Close immediate indicator, one bit.

lua_flag1.nowait

No wait for data flag, one bit.

lua_flag1.sscp_exp

SSCP expedited flow, one bit.

lua_flag1.sscp_norm

SSCP normal flow, one bit.

lua_flag1.lu_exp

LU expedited flow, one bit.

lua_flag1.lu_norm

LU normal flow, one bit.

lua_message_type

Specifies the type of the inbound or outbound SNA commands and data. This is a returned parameter for RUI_INIT and SLI_OPEN and a supplied parameter for SLI_SEND. For other LUA verbs this variable is not used and should be set to zero.

Possible values are:

LUA_MESSAGE_TYPE_LU_DATA

LUA_MESSAGE_TYPE_SSCP_DATA

LUA_MESSAGE_TYPE_BID

LUA_MESSAGE_TYPE_BIND

LUA_MESSAGE_TYPE_BIS

LUA_MESSAGE_TYPE_CANCEL

LUA_MESSAGE_TYPE_CHASE

LUA_MESSAGE_TYPE_CLEAR

LUA_MESSAGE_TYPE_CRV

LUA_MESSAGE_TYPE_LUSTAT_LU

LUA_MESSAGE_TYPE_LUSTAT_SSCP

LUA_MESSAGE_TYPE_QC

LUA_MESSAGE_TYPE_QEC

LUA_MESSAGE_TYPE_RELQ

LUA_MESSAGE_TYPE_RQR

LUA_MESSAGE_TYPE_RTR

LUA_MESSAGE_TYPE_SBI

LUA_MESSAGE_TYPE_SHUTD

LUA_MESSAGE_TYPE_SIGNAL

LUA_MESSAGE_TYPE_SDT

LUA_MESSAGE_TYPE_STSN

LUA_MESSAGE_TYPE_UNBIND

The SLI receives and responds to the BIND, CRV, and STSN requests through the LUA interface extension routines.

LU_DATA, LUSTAT_LU, LUSTAT_SSCP, and SSCP_DATA are not SNA commands.

lua_flag2

Returned parameter. Contains flags for messages returned by LUA. This parameter is returned by RUI_BID, RUI_READ, RUI_WRITE, SLI_BID, SLI_RECEIVE, and SLI_SEND. For other LUA verbs this parameter is not used and should be set to zero. Its subparameters are as follows:

lua_flag2.bid_enable

Indicates that **RUI_BID** was successfully re-enabled if set to 1.

lua_flag2.async

Indicates that the LUA interface verb completed asynchronously if set to 1.

lua_flag2.sscp_exp

Indicates SSCP expedited flow if set to 1.

lua_flag2.sscp_norm

Indicates SSCP normal flow if set to 1.

lua_flag2.lu_exp

Indicates LU expedited flow if set to 1.

lua_flag2.lu_norm

Indicates LU normal flow if set to 1.

lua_resv56

This supplied parameter is a reserved field used by SLI_OPEN and RUI_INIT. For all other LUA verbs, this parameter is reserved and should be set to zero.

lua_encr_decr_option

This parameter is a field for cryptography options. On **RUI_INIT**, only the following are supported:

- **lua_encr_decr_option** = 0
- **lua_encr_decr_option** = 128

For all other LUA verbs, this parameter is reserved and should be set to zero.

Values for lua_message_type

The following table describes the possible values for **lua_message_type**.

Message type	SNA data	SLI_SEND	SLI_BID SLI_RECEIVE	RUI_BID RUI_READ
0xC8	BID	X	X	X
0x31	BIND		Extension*	X
0x70	BIS	X	X	X
0x83	CANCEL	X	X	X
0x84	CHASE	X	X	X
0xA1	CLEAR			X
0xD0	CRV			X
0x01	LU_DATA**	X	X	X
0x04	LUSTAT_LU**	X	X	X
0x14	LUSTAT_SSCP**	X	X	X
0x81	QC	X	X	X
0x80	QEC	X	X	X
0x82	RELQ	X	X	X
0xA3	RQR	X		X
0x02	RSP	X	X	
0x05	RTR	X	X	X
0x71	SBI	X	X	X
0xC0	SHUTD			X
0xC9	SIGNAL	X	X	X
0xA0	SDT			X
0x11	SSCP_DATA**	X	X	X
0xA2	STSN		Extension*	X
0x32	UNBIND			X

*The SLI receives and responds to the BIND, CRV, and STSN requests through the LUA interface extension routines.

**Not an SNA command.

Command-Specific Structure of LUA VCBs

The following union shows the specific data structure that is included for functions that use the **LUA_SPECIFIC** part of a verb control block. The only LUA verbs that use this union are RUI_BID, SLI_BID, SLI_OPEN, and SLI_SEND.

```
union LUA_SPECIFIC {
    struct SLI_OPEN open;
    unsigned char lua_sequence_number[2];
    unsigned char lua_peek_data[12];
} LUA_SPECIFIC;
```

Members

open
The union member of **LUA_SPECIFIC** used by the **SLI_OPEN** verb.

lua_sequence_number
The union member of **LUA_SPECIFIC** used by the **SLI_SEND** verb. Returned parameter. Sequence number of the RU to the host. It contains the sequence number for either the first in the chain request unit or the only segment in the chain request unit. Note that this parameter is not byte-reversed.

lua_peek_data
The union member of **LUA_SPECIFIC** used by the **RUI_BID** and **SLI_BID** verbs. Returned parameter. Contains up to 12 bytes of the data waiting to be read. It is a preview (up to 12 bytes) of the RU data waiting to be read. The **lua_data_length** parameter contains the exact length of the data peeked at.

The following topic describes command-specific parameters for **SLI_OPEN**.

SLI_OPEN VCB Structure

The following structure shows the **SLI_OPEN** fields of the **LUA SPECIFIC** union member for the SLI_OPEN verb.

```
struct SLI_OPEN {
    unsigned char lua_init_type;
    unsigned char lua_resv65;
    unsigned short lua_wait;
    struct LUA_EXT_ENTRY lua_open_extension[3];
    unsigned char lua_ending_delim;
} SLI_OPEN;
```

Members

lua_init_type
Type of session initiation, which determines how the LU-LU session is initialized by the Windows LUA interface. The following values are valid:

lua_init_type_sec_is

Secondary-initiated and sends the INITSELF command supplied in the OPEN data buffer.

lua_init_type_sec_log

Secondary-initiated with an unformatted LOGON message in the OPEN data buffer.

lua_init_type_prim

Primary-initiated and waits on the BIND command.

lua_init_type_prim_sscp

Primary-initiated with SSCP access.

lua_resv65
Reserved field.

lua_wait
Secondary retry wait time. Specifies how many seconds the Windows LUA interface is to wait before retransmitting the INITSELF or the LOGON message after receiving one of the following:

- A NOTIFY command (indicating a procedure error)
- A network services procedure error message
- A negative response with one of the following secondary return codes:

  RESOURCE_NOT_AVAILABLE
  SESSION_LIMIT_EXCEEDED
  SESSION_SERVICE_PATH_ERROR

lua_open_extension
Supplied parameter. Specifies any user-supplied dynamic-link libraries (DLLs) used to process specific LUA messages.

lua_ending_delim
Extension list delimiter.

LUA Common Return Codes

This section describes the primary and, if applicable, secondary return codes that are common to the LUA verbs. The return codes are listed in hexadecimal order.

Verb-specific return codes are described for the individual verbs in LUA RUI Verbs and LUA SLI Verbs.

LUA Primary Return Codes

0x0000

LUA_OK
The verb executed successfully.

0x0001

LUA_PARAMETER_CHECK
The verb did not execute because of a parameter error.

0x0002

LUA_STATE_CHECK
The verb did not execute because it was issued in an invalid state.

0x000F

LUA_SESSION_FAILURE
A required Microsoft® Host Integration Server or Microsoft® SNA Server component (such as the local node) has terminated.

0x0014

LUA_UNSUCCESSFUL
The verb record supplied was valid, but the verb did not complete successfully.

0x0018

LUA_NEGATIVE_RESPONSE
Either LUA sent a negative response to a message received from the primary LU because an error was found in the message, or the application responded negatively to a chain for which the end-of-chain has arrived.

0x0021

LUA_CANCELED
The secondary return code gives the reason for canceling the command.

0x0030

LUA_IN_PROGRESS
An asynchronous command was received but is not completed.

0x0040

LUA_STATUS
The secondary return code contains SLI status information for the application.

0xF003

LUA_COMM_SUBSYSTEM_ABENDED
Indicates one of the following conditions:

- The node used by this conversation encountered an ABEND.
- The connection between the TP and the PU 2.1 node was broken (a LAN error).
- The SnaBase at the TP's computer encountered an ABEND.

0xF004

LUA_COMM_SUBSYSTEM_NOT_LOADED
A required component could not be loaded or terminated while processing the verb. Thus, communication could not take place. Contact the system administrator for corrective action.

0xF008

LUA_INVALID_VERB_SEGMENT
The VCB extended beyond the end of the data segment.

0xF011

LUA_UNEXPECTED_DOS_ERROR

After issuing an operating system call, an unexpected operating system return code was received and is specified in the secondary return code.

0xF015

LUA_STACK_TOO_SMALL
The stack size of the application is too small to execute the verb. Increase the stack size of your application.

0xFFFF

LUA_INVALID_VERB
Either the verb code or the operation code, or both, is invalid. The verb did not execute.

LUA Secondary Return Codes

0x00000000

LUA_SEC_RC_OK
No additional information exists for LUA_OK.

0x00000001

LUA_INVALID_LUNAME
An invalid **lua_luname** name was specified.

0x00000002

LUA_BAD_SESSION_ID
An invalid value for **lua_sid** was specified in the VCB.

0x00000003

LUA_DATA_TRUNCATED
The data was truncated because the data received was longer than the buffer length specified in **lua_max_length**.

0x00000004

LUA_BAD_DATA_PTR
The **lua_data_ptr** parameter either does not contain a valid pointer or does not point to a read/write segment and supplied data is required.

0x00000005

LUA_DATA_LENGTH_ERROR
One of the following occurred:

- The supplied data segment for **SLI_RECEIVE** or **SLI_SEND** is not a read/write data segment as required.
- The supplied data segment for **SLI_RECEIVE** is not as long as that provided in **lua_max_length**.
- The supplied data segment for **SLI_SEND** is not as long as that provided in **lua_data_length**.

0x00000006

LUA_RESERVED_FIELD_NOT_ZERO
A reserved parameter for the verb just issued is not set to zero.

0x00000007

LUA_INVALID_POST_HANDLE
For a Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 system using events as the asynchronous posting method, the Windows-based LUA VCB does not contain a valid event handle.

For the Windows version 3.*x* system, the LUA VCB does not contain the valid procedure address returned by the **MakeProcInstance** command.

For OS/2, the LUA VCB does not contain a valid semaphore or queue handle, which is needed when a verb completes asynchronously.

0x0000000C

LUA_PURGED
SLI_PURGE was issued and canceled **SLI_RECEIVE**.

0x0000000F

LUA_BID_VERB_ERROR
The buffer with the **SLI_BID** VCB was released before the **SLI_RECEIVE** with **lua_flag1.bid_enable** set to 1 was issued.

0x00000010

LUA_NO_PREVIOUS_BID_ENABLED
SLI_BID was not issued prior to issuing **SLI_RECEIVE** with **bid_enable**.

0x00000011

LUA_NO_DATA
No data was available to read when **SLI_RECEIVE** containing a no-wait parameter was issued.

0x00000012

LUA_BID_ALREADY_ENABLED
SLI_RECEIVE was issued with **bid_enable** when **SLI_BID** was already active.

0x00000013

LUA_VERB_RECORD_SPANS_SEGMENTS
The LUA VCB length parameter plus the segment offset is beyond the segment end.

0x00000014

LUA_INVALID_FLOW
The **lua_flag1** flow flags were set incorrectly when a verb was issued as follows:

- When issuing **SLI_SEND** to send an SNA response, set only one **lua_flag1** flow flag.
- When issuing **SLI_RECEIVE**, set at least one **lua_flag1** flow flag.

0x00000015

LUA_NOT_ACTIVE
LUA was not active within Microsoft® Host Integration Server or Microsoft® SNA Server when an LUA verb was issued.

0x00000016

LUA_VERB_LENGTH_INVALID
An LUA verb was issued with the value of **lua_verb_length** unexpected by the LUA.

0x00000019

LUA_REQUIRED_FIELD_MISSING
The verb that was issued either did not include a data pointer (if the data count was not zero) or did not include an **lua_flag1** flow flag.

0x00000030

LUA_READY
Following a NOT_READY status, this status is issued to notify you that the SLI is ready to process commands.

0x00000031

LUA_NOT_READY
One of the following caused the SLI session to be temporarily suspended:

- An SNA UNBIND type 0x02 command was received, indicating a new BIND is coming.

  If the UNBIND type 0x02 is received after the beginning **SLI_OPEN** is complete, the session is suspended until a BIND, optional CRV and STSN, and SDT flows are received. These routines are re-entrant because they have to be called again. The session resumes after the SLI processes the SDT command. If the UNBIND type 0x02 is received while **SLI_OPEN** is still processing, the primary return code is session-failure, not status.

- The receipt of an SNA CLEAR caused the suspension.

  Receipt of an SNA SDT will cause the session to resume.

0x00000032

LUA_INIT_COMPLETE
The LUA interface initialized the session while **SLI_OPEN** was processing. LUA applications that issue **SLI_OPEN** with **lua_open_type_prim_sscp** receive this status on **SLI_RECEIVE** or **SLI_BID**.

0x00000033

LUA_SESSION_END_REQUESTED
The LUA interface received an SNA shutdown command (SHUTD) from the host, indicating the host is ready to shut down the session.

0x00000034

LUA_NO_SLI_SESSION
A session was not open or was down due to an **SLI_CLOSE** or session failure when a command was issued.

0x00000035

LUA_SESSION_ALREADY_OPEN
A session is already open for the LU name specified in **SLI_OPEN**.

0x00000036

LUA_INVALID_OPEN_INIT_TYPE
The value in the **lua_init_type** contained in **SLI_OPEN** is invalid.

0x00000037

LUA_INVALID_OPEN_DATA
The **lua_init_type** for the **SLI_OPEN** issued is set to LUA_INIT_TYPE_SEC_IS when the buffer for data does not have a valid INITSELF command.

0x00000038

LUA_UNEXPECTED_SNA_SEQUENCE
Unexpected data or commands were received from the host while **SLI_OPEN** was processing.

0x00000039

LUA_NEG_RSP_FROM_BIND_ROUTINE
The user-supplied SLI_BIND routine responded negatively to the BIND. **SLI_OPEN** ended unsuccessfully.

0x0000003B

LUA_NEG_RSP_FROM_STSN_ROUTINE
The user-supplied SLI STSN routine responded negatively to the STSN. **SLI_OPEN** ended unsuccessfully.

0x0000003E

LUA_INVALID_OPEN_ROUTINE_TYPE
The **lua_open_routine_type** for the **SLI_OPEN** list of extension routines is invalid.

0x0000003F

LUA_MAX_NUMBER_OF_SENDS
The application issued a third **SLI_SEND** before one completed.

0x00000040

LUA_SEND_ON_FLOW_PENDING
An **SLI_SEND** was still outstanding when the application issued another **SLI_SEND** for an SNA flow.

0x00000041

LUA_INVALID_MESSAGE_TYPE
The **lua_message_type** parameter is not recognized by the LUA interface.

0x00000042

LUA_RECEIVE_ON_FLOW_PENDING
An **SLI_RECEIVE** was still outstanding when this application issued another **SLI_RECEIVE** for an SNA flow.

0x00000043

LUA_DATA_LENGTH_ERROR
The application did not provide user-supplied data required by the verb issued. Note that when **SLI_SEND** is issued for an SNA LUSTAT command, status (in four bytes) is required, and that when **SLI_OPEN** is issued with secondary initialization, data is required.

0x00000044

LUA_CLOSE_PENDING
One of the following occurred:

- A CLOSE_ABEND was still pending when another CLOSE_ABEND was issued. You can issue a CLOSE_ABEND if a CLOSE_NORMAL is pending.

- Either a CLOSE_ABEND or a CLOSE_NORMAL was still pending when a CLOSE_NORMAL was issued.

0x00000046

LUA_NEGATIVE_RSP_CHASE
A negative response to an SNA CHASE command from the host was received by the LUA interface while **SLI_CLOSE** was being processed. **SLI_CLOSE** continued processing to stop the session.

0x00000047

LUA_NEGATIVE_RSP_SHUTC
A negative response to an SNA SHUTC command from the host was received by the SLI while **SLI_CLOSE** was still being processed. **SLI_CLOSE** continued processing to stop the session.

0x00000048

LUA_NEGATIVE_RSP_RSHUTD
A negative response to an SNA RSHUTD command from the host was received by the LUA interface while **SLI_CLOSE** was being processed. **SLI_CLOSE** continued processing to stop the session.

0x0000004A

LUA_NO_RECEIVE_TO_PURGE
No **SLI_RECEIVE** was outstanding when you issued **SLI_PURGE**. One of two situations caused the problem:

- **SLI_RECEIVE** completed before **SLI_PURGE** finished processing.

  You can change the application to take care of this problem because it is not an error condition.

- The **lua_data_ptr** parameter does not correctly point to the **SLI_RECEIVE** you want to purge.

0x0000004D

LUA_CANCEL_COMMAND_RECEIVED
The host sent an SNA CANCEL command to cancel the data chain currently being received by **SLI_RECEIVE**.

0x0000004E

LUA_RUI_WRITE_FAILURE
An unexpected error was posted to the SLI by **RUI_WRITE**.

0x00000051

LUA_SLI_BID_PENDING
An SLI verb was still active when another **SLI_BID** was issued. Only one **SLI_BID** can be active at a time.

0x00000052

LUA_SLI_PURGE_PENDING
An **SLI_PURGE** was still active when another **SLI_PURGE** was issued. Only one **SLI_PURGE** can be active at a time.

0x00000053

LUA_PROCEDURE_ERROR
A host procedure error is indicated by the receipt of an NSPE or NOTIFY message. The return code is posted to **SLI_OPEN** when the retry option is not used. To use the reset option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue **SLI_CLOSE**.

0x00000054

LUA_INVALID_SLI_ENCR_OPTION
The **lua_encr_decr_option** parameter was set to 128 in **SLI_OPEN**, which is not supported for the encryption/decryption processing option.

0x00000055

LUA_RECEIVED_UNBIND
The primary LU sent an SNA UNBIND command to the LUA interface when a session was active. As a result, the session was stopped.

0x0000007F

LUA_SLI_LOGIC_ERROR

The LUA interface found an internal error in logic.

0x00000080

LUA_TERMINATED
The session was terminated when a verb was pending. The verb process has been canceled.

0x00000081

LUA_NO_RUI_SESSION
No session has been initialized for the LUA verb issued, or some verb other than **SLI_OPEN** was issued before the session was initialized.

0x00000083

LUA_INVALID_PROCESS
The session for which an RUI verb was issued is unavailable because another process owns the session.

0x0000008C

LUA_LINK_NOT_STARTED
The LUA was not able to activate the data link during initialization of the session.

0x0000008D

LUA_INVALID_ADAPTER
The configuration for the DLC is in error, or the configuration file is corrupted.

0x0000008E

LUA_ENCR_DECR_LOAD_ERROR
An unexpected return code was received from the OS/2 **DosLoadModule** function while attempting to load the user-provided encryption or decryption dynamic link module.

0x0000008F

LUA_ENCR_DECR_LOAD_ERROR
An unexpected return code was received from the OS/2 **DosGetProcAddr** function while attempting to get the procedure address within the user-provided encryption or decryption dynamic link module.

0x000000BE

LUA_NEG_NOTIFY_RSP
The SSCP responded negatively to a NOTIFY request issued indicating that the secondary LU was capable of a session. The half-session component that received the request understood and supported the request but could not execute it.

0x000000FF

LUA_LU_INOPERATIVE
A severe error occurred while the RUI was attempting to stop the session. This LU is unavailable for any LUA requests until an ACTLU is received from the host.

0x08010000

LUA_RESOURCE_NOT_AVAILABLE
The logical unit, physical unit, link, or link station specified in the request unit is unavailable. This return code is posted to **SLI_OPEN** when a resource is unavailable unless you use the retry option.

To use the retry option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue **SLI_CLOSE**.

0x08050000

LUA_SESSION_LIMIT_EXCEEDED
The session requested was not activated because an NAU is at its session limit.

This SNA sense code applies to the following requests: BID, CINIT, INIT, and ACTDRM. The code will be posted to **SLI_OPEN** when an NAU is at its limit, unless you use the retry option.

To use the retry option, set **lua_wait** to a value other than zero. The LOGON or INITSELF command will be retried until the host is ready or until you issue **SLI_CLOSE**.

0x08090000

LUA_MODE_INCONSISTENCY
Performing this function is not allowed by the current status. The request sent to the half-session component was not executed even though it was understood and supported. This SNA sense code is also an exception request sense code.

0x08120000

LUA_INSUFFICIENT_RESOURCES
A temporary condition of insufficient resources caused the request receiver to be unable to perform. The request sent to the half-session component was not executed, even though it was understood and supported.

0x081B0000

LUA_RECEIVER_IN_TRANSMIT_MODE
Either resources needed to handle normal flow data were not available or the state of the half-duplex contention was not received when a normal-flow request was received. The result is a race condition. This SNA sense code is also an exception request sense code.

0x08310000

LUA_LU_COMPONENT_DISCONNECTED
An LU component is unavailable because it is not connected properly. Make sure that the power is on.

0x08350001

LUA_NEGOTIABLE_BIND_ERROR
A negotiable BIND was received, which is only allowed by the SLI when a user-supplied SLI_BIND routine is provided with **SLI_OPEN**.

0x08350002

LUA_BIND_FM_PROFILE_ERROR
Only file management header profiles 3 and 4 are supported by the LUA interface. A file management profile other than 3 or 4 was found on the BIND.

0x08350003

LUA_BIND_TS_PROFILE_ERROR
Only TS profiles 3 and 4 are supported by the LUA interface. A TS other than 3 or 4 was found on the BIND.

0x0835000E

LUA_BIND_LU_TYPE_ERROR
Only LU 0, LU 1, LU 2, and LU 3 are supported by LUA. An LU other than 0, 1, 2, or 3 was found.

0x08570000

LUA_SSCP_LU_SESSION_NOT_ACTIVE
The required SSCP-LU is inactive. Specific sense code information is in bytes 2 and 3. Valid settings are 0x0000, 0x0001, 0x0002, 0x0003, and 0x0004.

0x08780001

LUA_RECEIVE_CORRELATION_TABLE_FULL
The session receive correlation table for the flow requested reached its capacity.

0x08780002

LUA_SEND_CORR_TABLE_FULL
The session send correlation table for the flow requested reached its capacity.

0x087D0000

LUA_SESSION_SERVICE_PATH_ERROR
A request for session services cannot be rerouted to an SSCP-SSCP session path. Specific sense code information in bytes 2 and 3 gives more information about why the request cannot be rerouted.

0x10020000

LUA_RU_LENGTH_ERROR
The RU request was an incorrect length (either too short or too long). The request unit was not interpreted or processed even though it was delivered to the half-session component. The half-session capabilities do not match. This SNA sense code is also an exception request sense code.

0x10030000

LUA_FUNCTION_NOT_SUPPORTED
The LUA does not support the requested function. A control character, an RU parameter, or a formatted request code may have specified the function. Specific sense code information is in bytes 2 and 3.

0x10050121

LUA_HDX_BRACKET_STATE_ERROR
The existing state error prevented the current request from being sent. The determination was made by a protocol computer.

0x10050122

LUA_RESPONSE_ALREADY_SENT
A response for the chain was already sent so that the current request was not sent. The determination was made by a protocol computer.

0x10050123

LUA_EXR_SENSE_INCORRECT
The application responded negatively to an exception request. The sense code was unacceptable.

0x10050124

LUA_RESPONSE_OUT_OF_ORDER
The current response was not for the oldest request. The determination was made by a protocol computer.

0x10050125

LUA_CHASE_RESPONSE_REQUIRED
A CHASE response was still outstanding when a more recent request was attempted. The determination was made by a protocol computer.

0x20020000

LUA_CHAINING_ERROR
The sequence of the chain indicator settings is in error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20030000

LUA_BRACKET
The sender failed to enforce the session bracket rules. Note that contention and race conditions are exempt from this error. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20040000

LUA_DIRECTION
While the half-duplex flip-flop state was NOT_RECEIVE, a request for normal flow was received. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20050000

LUA_DATA_TRAFFIC_RESET
A half-session of an active session with inactive data traffic received a normal flow DFC or FMD request. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20060000

LUA_DATA_TRAFFIC_QUIESCED
A DFC or FMD request was received from a half-session that sent either a SHUTC command or QC command, and the DFC or FMD request has not responded to a RELQ command. An invalid request header or request unit for the receiver's current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20070000

LUA_DATA_TRAFFIC_NOT_RESET
While the data traffic state was not reset, the session control request was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was

prevented.

0x20080000

LUA_NO_BEGIN_BRACKET
The receiver has already sent a positive response to a BIS command when a BID or an FMD request specifying BBI=BB was received. An invalid request header or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x20090000

LUA_SC_PROTOCOL_VIOLATION
A violation of the SC protocol occurred. A request (that is permitted only after an SC request and a positive response to that request have been successfully exchanged) was received before the required exchange. Byte 4 of the sense data contains the request code. No user data exists for this sense code. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x200A0000

LUA_IMMEDIATE_REQUEST_MODE_ERROR
The request violated the immediate request mode protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x200B0000

LUA_QUEUED_RESPONSE_ERROR
The request violated the queued response protocol. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x200C0000

LUA_ERP_SYNC_EVENT_ERROR
A violation of the ERP synchronous event protocol occurred. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x200D0000

LUA_RSP_BEFORE_SENDING_REQ
A previously received request has not yet been responded to and an attempt was made in half-duplex send/receive mode to send a normal flow request. An invalid header request or request unit for the received current session control or data flow control state was found. Delivery to the half-session component was prevented.

0x200E0000

LUA_RSP_CORRELATION_ERROR
A response was sent that does not correspond to a previously received request or a response was received that does not correspond to a request sent previously.

0x200F0000

LUA_RSP_PROTOCOL_ERROR
A violation of the response protocol was found in the response received from the primary half-session.

0x40010000

LUA_INVALID_SC_OR_NC_RH
The RH of an SC or NC request was invalid.

0x40030000

LUA_BB_NOT_ALLOWED
The begin bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40040000

LUA_EB_NOT ALLOWED
The end bracket indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40060000

LUA_EXCEPTION_RSP_NOT_ALLOWED
When an exception response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40070000

LUA_DEFINITE_RSP_NOT_ALLOWED
When a definite response was not allowed, one was requested. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40080000

LUA_PACING_NOT_SUPPORTED
The request contained a pacing indicator when support of pacing for this session does not exist for the receiving half-session or boundary function half-session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40090000

LUA_CD_NOT_ALLOWED
The change-direction indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x400A0000

LUA_NO_RESPONSE_NOT_ALLOWED
A request other than an EXR contained a NO RESPONSE. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x400B0000

LUA_CHAINING_NOT_SUPPORTED
The chaining indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x400C0000

LUA_BRACKETS_NOT_SUPPORTED
The bracket indicators were incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x400D0000

LUA_CD_NOT_SUPPORTED
The change-direction indicator was set, but LUA does not support change-direction for this situation. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x400F0000

LUA_INCORRECT_USE_OF_FI
The format indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40100000

LUA_ALTERNATE_CODE_NOT_SUPPORTED
The code selection indicator was set, but LUA does not support code selection for this session. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session

component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40110000

LUA_INCORRECT_RU_CATEGORY
The request unit category indicator was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40120000

LUA_INCORRECT_REQUEST_CODE
The request code was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40130000

LUA_INCORRECT_SPEC_OF_SDI_RTI
The SDI and the RTI were not specified correctly on a response. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40140000

LUA_INCORRECT_DR1I_DR2I_ERI
The DR1I, the DR2I, and the ERI were specified incorrectly. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40150000

LUA_INCORRECT_USE_OF QRI
The QRI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40160000

LUA_INCORRECT_USE_OF_EDI
The EDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x40170000

LUA_INCORRECT_USE_OF_PDI
The PDI was incorrectly specified. The BIND options chosen previously or the architectural rules were violated by the request header parameter values. Delivery to the half-session component was prevented. The errors are not dependent on the current session state. The sender's failure to enforce session rules may have caused the errors.

0x80030000

LUA_NAU_INOPERATIVE
The NAU is not able to process responses or requests. Delivery to the receiver could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, that usually means there is no longer a valid path to the session partner.

0x80050000

LUA_NO_SESSION
A request to activate a session is required because no active half-session in the receiving end node for the origination-destination pair exists, or no active boundary function half-session component for the origination-destination pair in a node that supplies the boundary function exists. Delivery of the request could not take place for one of the following reasons:

- A path information unit error
- A path outage
- An invalid sequence of requests for activation

If a path error is received during an active session, that usually indicates there is no longer a valid path to the session partner.

LUA Sample Applications

This section contains descriptions of samples that show how to build a simple 3270 emulator using either the RUI or SLI API.

This section contains:

- LUA Code Samples in the SDK

LUA Code Samples in the SDK

The source code for several sample programs that illustrate using LUA are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\SNA subdirectory on the Host Integration Server 2000 CD-ROM (these samples are located under the \SDK\SAMPLES folder on earlier versions of SNA Server). These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\SNA subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\Sna subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files:

Sample TP program	**Description**
RUI3270	Sample code for a simple emulator based on using the RUI API. Sample code is provided for Win32 and Win16 applications. This sample is located in the \SDK\Samples\SNA\RUI3270 folder on the CD-ROM.
SLI3270	Sample code for a simple emulator based on using the SLI API. Sample code is provided for Win32 and Win16 applications. This sample is located in the \SDK\Samples\SNA\SLI3270 folder on the CD-ROM.

These samples show how to build a simple 3270 emulator using either the RUI or SLI API. This emulator does not interpret the data it receives from the host, but does demonstrate how to use the APIs to establish a session with the host.

Building the LUA Samples

The LUA samples are designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio .NET interactive development environment (IDE).

To build the LUA samples installed from the Host Integration Server CD-ROM, set the following environment variables:

Variable	Description
ISVLIBS	The directory containing the Microsoft® Host Integration Server 2000 LIB files for Microsoft® Windows 2000, Microsoft® Window NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95.
ISVINCS	The directory containing the Host Integration Server 2000 header files.
SAMPLEROOT	The root directory where the sample code provided as part of the SDK has been installed on a local hard disk.

For example, if you installed the Host Integration Server SDK directory to the default location (C:\Program Files\Host Integration Server SDK), use the following lines to set the variables (assumes Intel binaries are being produced for Windows 2000, Windows NT on I386, Windows 98, or Windows 95) :

```
ISVLIBS=C:\Program Files\Host Integration Server SDK\LIB
ISVINCS=C:\Program Files\Host Integration Server SDK\Include
SAMPLEROOT=C:\Program Files\Host Integration Server SDK\Samples\SNA
```

Change to each subdirectory and run NMAKE on the .MAK file in each directory. For example, for the Win32 version of Rui3270, change to the SNA\Rui3270\win32 directory and type the following:

nmake -f nrui3270.mak

Note that Windows NT on DEC Alpha is not supported by the Host Integration Server SDK. If you wish to build these samples on Windows NT 4.0 for DEC Alpha, the earlier SNA Server 4.0 SDK will be required for accessing the Windows NT import libraries for DEC Alpha under the \SDK\LIB\WINNT\ALPHA folder.

To build the LUA samples installed as part of the MSDN Platform SDK using the command-line compiler, set up your build environment as follows:

- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory. The default location of this file is C:\Program Files\Microsoft Visual Studio\VC98\Bin (for VS6) or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools (for VS.NET)

To build all the SNA samples, open an MS-DOS Command Prompt window, navigate to the SNA subdirectory, and invoke NMAKE. This will recursively invoke NMAKE and build all of the SNA samples including the LUA samples.

To build a specific sample (Rui3270, for example) using the command-line compiler, open an MS-DOS Command Prompt window, navigate to the appropriate subdirectory (SNA\Rui3270\win32, for example), and invoke NMAKE.

To build a specific sample (the Win32 version of Rui3270, for example) using the Visual Studio .NET IDE, start Microsoft Visual Studio. NET 7.0 and open the appropriate Visual C++ 7.0 project file (SNA\Rui3270\win32\nrui3270.vcproj, for example) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

Specifying a File Name for Table G for Code Conversion

The sample emulators use a user-defined table referred to as Table G in Microsoft® Host Integration Server or Microsoft® SNA Server for converting between ASCII and EBCDIC characters. The sample applications require that the file name of this table be specified. Use the COMTBLG registry or create a COMTBLG environment variable to specify the file name. The registry entry is described in the sections on Host Integration Server Client Binary Setup.

A sample Table G file, COMTBLG.DAT, is installed with Host Integration Server or SNA Server 4.0 in the SYSTEM subdirectory below the root directory where the product is installed. The default location where Host Integration Server is installed is the following:

C:\Program Files\Host Integration Server

To use this COMTBLG.DAT file, copy it to the client computers where the sample programs will be run, and specify the file name using the COMTBLG environment variable or the registry entry.

Code Samples Using the RUI API

The following files located under the SNA\RUI3270 folder can be used to build simple 3270 emulators using the RUI API. Study the comments in the .C files for information about how to run the emulators.

File name	API Used	Type of file	Target operating system
RUI3270.C	RUI	Source code for RUI3270.EXE	Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, OS/2, or Microsoft® MS-DOS®
Win32\NRUI3270.MAK	RUI	Makefile	Windows 2000, Windows NT, Windows 98, or Windows 95
Win32\RUIINC.C	RUI	Source code that #includes RUI3270.c to build a Win32 version of WRUI3270.EXE	Windows 2000, Windows NT, Windows 98, or Windows 95
Win32\WRUI3270.RC	RUI	Resource definition file	Windows 2000, Windows NT, Windows 98, or Windows 95
Win16\WRUI3270.C	RUI	Source code for WRUI3270.EXE	Windows version 3.1 or Windows for Workgroups
Win16\WRUI3270.DEF	RUI	Definition file	Windows version 3.1 or Windows for Workgroups
Win16\WRUI3270.MAK	RUI	Makefile	Windows version 3.1 or Windows for Workgroups
Win16\WRUI3270.RC	RUI	Resource definition file	Windows version 3.1 or Windows for Workgroups

Note that the earlier SNA Server SDK also included makefiles for building OS/2 and MS-DOS versions of this sample.

Code Samples Using the SLI API

The following files located under the SNA\SLI3270 folder can be used to build simple 3270 emulators using the SLI API. Study the comments in the .C files for information about how to run the emulators.

File name	API Used	Type of file	Target operating system
SLI3270.C	SLI	Source code for SLI3270.EXE	Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, Microsoft® Windows® 95, OS/2, or Microsoft® MS-DOS®
Win32\NSLI3270.MAK	SLI	Makefile	Windows 2000, Windows NT, Windows 98, or Windows 95
Win32\SLIINC.C	SLI	Source code that #includes SLI3270.c to build a Win32 version of SLI3270.EXE	Windows 2000, Windows NT, Windows 98, or Windows 95
Win32\WSLI3270.RC	SLI	Resource definition file	Windows 2000, Windows NT, Windows 98, or Windows 95
Win16\WSLI3270.C	SLI	Source code for WSLI3270.EXE	Windows version 3.1 or Windows for Workgroups
Win16\WSLI3270.DEF	SLI	Definition file	Windows version 3.1 or Windows for Workgroups
Win16\WSLI3270.MAK	SLI	Makefile	Windows version 3.1 or Windows for Workgroups
Win16\WSLI3270.RC	SLI	Resource definition file	Windows version 3.1 or Windows for Workgroups

Note that the earlier SNA Server SDK also included a makefile for building an OS/2 version of this sample.

3270 Emulator Interface Specifications

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information for independent software vendors who are developing their own 3270 emulation client software to work with Microsoft Host Integration Server 2000 or Microsoft SNA Server.

This section contains:

About the EIS Guide

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information for independent software vendors who are developing their own 3270 emulation client software to work with Microsoft® Host Integration Server 2000 or Microsoft SNA Server.

To use this section of the guide effectively, you should be familiar with:

- Microsoft Host Integration Server 2000
- One of the following operating environments:
    - Microsoft Windows® 2000
    - Microsoft Windows NT®
    - Microsoft Windows 98
    - Microsoft Windows 95
- SNA concepts

Before using this section, you should be familiar with System Network Architecture (SNA) concepts. This section provides the following information:

- Internal concepts of Host Integration Server 2000 and SNA Server that are required to integrate 3270 client software.
- Definitions of the interfaces used by the client software to communicate with Host Integration Server 2000 or SNA Server components.
- Information on using configuration and diagnostics features of Host Integration Server 2000 or SNA Server.
- Instructions for compiling and linking the client software with the necessary library files supplied with Host Integration Server 2000 or SNA Server.

This section contains:

Operating Systems Support for 3270 Development

This section of the guide contains information relating to following operating systems:

- Microsoft Windows 2000
- Microsoft Windows NT
- Microsoft Windows 98
- Microsoft Windows 95
- Microsoft Windows version 3.*x*
- Microsoft MS-DOS®
- OS/2

Microsoft Host Integration Server 2000 supports the development of 3270 client applications for Windows 2000, Windows NT, Windows 98, and Windows 95. Under these operating systems, support for 3270 client applications is provided only for the Win32® subsystem.

The previous Microsoft SNA Server product also supported the development of 3270 client applications for Windows 3.*x*, MS-DOS, and OS/2. Most 3270 client applications developed for Windows 3.*x*, MS-DOS, and OS/2 with SNA Server can be used with Host Integration Server 2000. The Windows 3.x, MS-DOS, and OS/2 interface is described here for completeness, but Windows 3.x, MS-DOS, or OS/2 3270 client application development is not supported using Host Integration Server.

Information presented here can be different for different operating systems. When a section of text applies only to specific operating systems, this is indicated by a header preceding that text. The end of such a section is indicated by one of the following:

- A header specifying other operating systems.
- A header stating that subsequent information applies to all operating systems.
- The end of a topic.

Text not explicitly designated applies to all operating systems.

Network Operating Systems Support for 3270 Development

Microsoft Host Integration Server 2000 supports the following network operating systems:

- Native TCP/IP
- Novell NetWare
- Banyan VINES (only when installed on Windows NT)

The network operating systems supported by Microsoft SNA Server version 4.0 are Microsoft LAN Manager, IBM LAN Server, Novell NetWare, native TCP/IP, and Banyan VINES.

3270 Emulation Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about how to use the 3270 Emulator to assist in writing applications for Host Integration Server 2000.

This section contains:

Host Integration Server Concepts

This section describes some key concepts used in Microsoft® Host Integration Server 2000 and Microsoft SNA Server when providing 3270 client access. Since the purpose of this document is to enable independent software vendors to integrate their 3270 emulators with a Host Integration Server or SNA Server system, only the relevant parts of the Microsoft SNA gateway architecture are described.

This section contains:

Structure of Host Integration Server Components

The components of Microsoft Host Integration Server 2000 and Microsoft SNA Server are local nodes, link services, the 3270 emulation program, and so on. This section introduces the structure of these components and explains terms used to refer to the structure.

This section contains:

The Role of the Base

The Base is a part of each Host Integration Server or SNA Server gateway component, such as the local 2.1 node or a link service, that provides the operating environment for the core functions of that component. It passes messages between components and provides functions common to all components, such as diagnostic tracing.

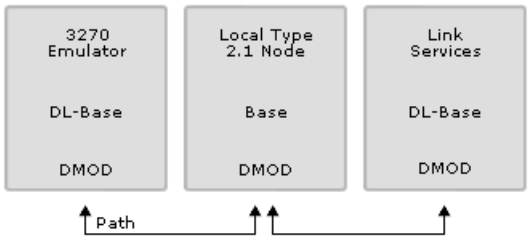
This document is concerned with the DL-BASE, which is the type of Base used by Host Integration Server and SNA Server 3270 emulation programs. The Host Integration Server or SNA Server DL-BASE supports a single SNA server component or a single user application and has entry points for initialization, sending messages, receiving messages, and termination.

Localities and DMODs

A Base and the components within it (that is, a Host Integration Server or an SNA Server executable program) is called a locality. The Host Integration Server or SNA Server system therefore consists of one or more communicating localities (all the running SNA server executable programs within the LAN Manager domain). For each SNA server system, there is a single configuration file.

In a system such as Host Integration Server, where the number of localities and their types are not configured in advance, the relationships between the localities are set up dynamically as individual localities come and go. Localities that can enter and leave a system in this way are called dynamic localities.

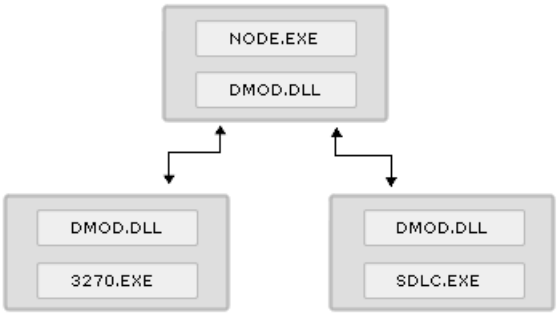
Dynamic localities communicate using the DMOD (dynamic access module) component, which provides the communications facilities needed to pass messages between the Bases. This is illustrated in the following figure.



This diagram shows a system consisting of three dynamic localities. Dynamic localities can enter or leave this system at any time.

For Windows® 2000, Windows NT®, Windows 98, Windows 95, Windows 3.*x*, and OS/2

The DMOD is implemented as a dynamic-link library (DLL). The preceding diagram can therefore be represented as follows:



Application Localities

Applications such as 3270 emulators can enter dynamically into an SNA server system. The application, in conjunction with the Base, acts as a whole locality and communicates with the other localities in the system using a DMOD.

The DL-BASE/DMOD Interface describes the interface to the Base and the DMOD that allows an application to participate in an SNA server system.

Partners

For Host Integration Server or SNA Server components and applications to communicate with each other, it must be possible to identify a partner within a locality. A partner is an addressable component of a locality; that is, code to which messages can be sent. In a Host Integration Server or SNA Server system, there is generally only one partner within a locality (such as a link service or the 3270 emulation program); however, separate functions within the local 2.1 node (such as the 3270 and APPC functions) can be considered to be separate partners.

Messages

Messages are used to pass data between partners in the Microsoft Host Integration Server 2000 or Microsoft SNA Server system. This section provides information about message structure and formats.

This section contains:

Overview of Message Formats

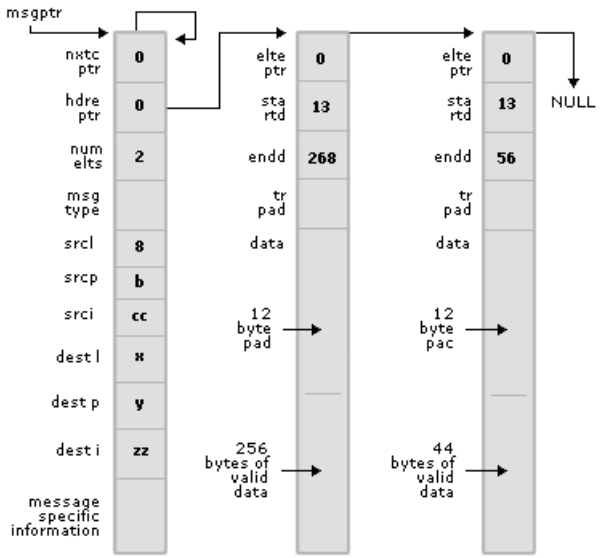
A message always contains fixed-format header information such as a message type and addressing information. It can also contain other header information specific to a particular message type (such as the message subtype) and an indefinite amount of extra data.

Messages are saved in buffers that consist of one header and zero or more elements:

- The header contains the fixed-format information and a pointer to an element. (This pointer will be NULL if there are no elements associated with the message.)
- An element contains any extra data for a message and a pointer to another element if the data continues into another element.

Buffer headers and elements are regarded as contiguous (8-bit) byte sequences. Messages of any length can be built by chaining sufficient elements to a header.

The following figure illustrates a typical message with two elements. The individual fields in the header and elements are explained in the following topics.



Buffer Header Format

This topic lists the common fields that always occur at the start of a buffer header. These are followed by further fields specific to the particular message; see FMI Message Formats for details of individual message formats.

Field	Type	Description
nxtqptr	PTRBFHDR	When the buffer is in a queue, this field points to the header of the next buffer in the queue (NULL if it is the last buffer in the queue). When the buffer is not in a queue, this field points to itself; the SNA server buffer management routines use this to check for buffer corruption.
hdreptr	PTRBFELT	Pointer to the first buffer element in the associated chain of buffer elements; NULL if the message consists only of a buffer header.
numelts	CHAR	Number of buffer elements chained from the header; zero if the message consists only of a buffer header.
msgtype	CHAR	Message type: see individual message descriptions in FMI Message Formats.
srcl	CHAR	Source locality: see LPI Addresses.
srcp	CHAR	Source partner: see LPI Addresses.
srci	INTEGER	Source index: see LPI Addresses.
destl	CHAR	Destination locality: see LPI Addresses.
destp	CHAR	Destination partner: see LPI Addresses.
desti	INTEGER	Destination index: see LPI Addresses.

```
PTRBFHDR  nxtqptr;
PTRBFELT  hdreptr;
CHAR      numelts;
CHAR      msgtype;
CHAR      srcl;
CHAR      srcp;
INTEGER   srci;
CHAR      destl;
CHAR      destp;
INTEGER   desti;
};
```

Members

nxtqptr
When the buffer is in a queue, this field points to the header of the next buffer in the queue (NULL if it is the last buffer in the queue). When the buffer is not in a queue, this field points to itself; the SNA server buffer management routines use this to check for buffer corruption.

hdreptr
Pointer to the first buffer element in the associated chain of buffer elements; NULL if the message consists only of a buffer header.

numelts
Number of buffer elements chained from the header; zero if the message consists only of a buffer header.

msgtype
Message type. See individual message descriptions in FMI Message Formats.

srcl
Source locality. See LPI Addresses.

srcp
Source partner. See LPI Addresses.
srci
Source index. See LPI Addresses.
destl
Destination locality. See LPI Addresses.
destp
Destination partner. See LPI Addresses.
desti
Destination index. See LPI Addresses.

Note Fields that occupy two bytes, such as **opresid** in Open(PLU) Request are normally represented with the arithmetically most significant byte in the lowest byte address, irrespective of the normal orientation used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. However, the following fields are exceptions:

- The **srci** and **desti** fields in buffer headers are stored in the local format of the application that assigns them (only the assigning application needs to interpret these values).
- The startd and endd fields in elements are always stored in low-byte, high-byte orientation (the normal orientation of an Intel processor).

Buffer Element Format

This topic lists the common fields that always occur at the start of a buffer element. The **dataru** field contains information specific to the particular message; see FMI Message Formats for details of individual message formats.

Field	Type	Description
hdreptr->elteptr	PTRBFELT	Pointer to next buffer element in the chain; NULL if this element is the last or only element in the chain.
hdreptr->startd	INTEGER	Start of valid data in this element. The index into **dataru** of the first byte of valid data.
hdreptr->endd	INTEGER	End of valid data in this element. The index into **dataru** of the last byte of valid data.
hdreptr->trpad	CHAR	Pad byte (reserved).
hdreptr->dataru	CHAR[268]	An array of characters that contains the data for this element. Note that the valid data might not occupy the whole of the element; **startd** and **endd** (see above) give the indexes into this array of the start and end of the valid data.

```
PTRBFELT   hdreptr->elteptr;
INTEGER    hdreptr->startd;
INTEGER    hdreptr->endd;
CHAR       hdreptr->trpad;
CHAR[268]  hdreptr->dataru;
};
```

Members

hdreptr–>elteptr
Pointer to next buffer element in the chain; NULL if this element is the last or only element in the chain.

hdreptr–>startd
Start of valid data in this element. The index into **dataru** of the first byte of valid data.

hdreptr–>endd
End of valid data in this element. The index into **dataru** of the last byte of valid data.

hdreptr–>trpad
Pad byte (reserved).

hdreptr–>dataru
An array of characters that contains the data for this element. Note that the valid data might not occupy the whole of the element; **startd** and **endd** give the indexes into this array of the start and end of the valid data.

The following information will help you to interpret the message formats:

- Certain messages are shown as having two elements in the message formats; for example, the Open(PLU) Request has the CICB field in the first element and the BIND RU in the second element. This indicates that the message consists of two distinct linked element chains; the **elteptr** field in the first element points to the second element.
- Fields that occupy two bytes are represented with the arithmetically most significant byte in the lowest byte address, irrespective of the normal orientation used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. The exceptions to this are the startd and endd fields in elements, which are always stored in low-byte, high-byte orientation (the normal orientation of an Intel processor).
- The offsets indicated by the startd and endd fields are expressed in terms of the first byte of dataru being offset 1; the first byte of valid data is at dataru[startd–1]. For example, if startd is 11 and endd is 18, then dataru begins with 10 bytes that are not valid data, followed by 8 bytes of valid data.
- It is possible for an element to arrive with startd greater than endd. This indicates there is no valid data in dataru.

In the sample message format illustrated in Overview of Message Formats, each element has a **startd** of 13, indicating 12 bytes of padding before the start of the valid data. This leaves room for 256 bytes of data, and hence the element data (300 bytes long in this example) requires two elements.

LPI Connections

Partners communicate by passing messages to each other. If two partners wish to communicate with each other, an LPI connection is set up between the two partners. Messages then flow between the partners over this connection. The term "LPI connection" is explained in LPI Addresses; note that this is not related to the Microsoft Host Integration Server or Microsoft SNA Server concept of a connection between the local node and a remote system.

This section contains:

- Paths and DMODs
- LPI Addresses
- Making Connections

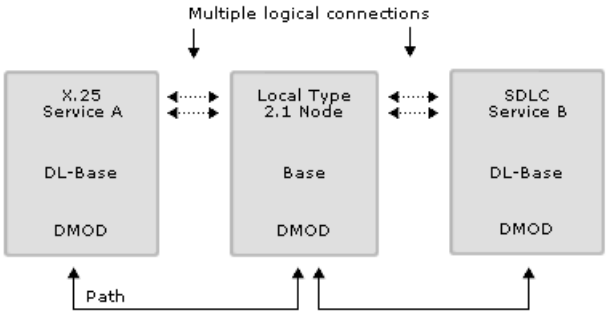
Paths and DMODs

Dynamic access modules (DMODs) are responsible for the communication between localities. When the DMODs in two localities can successfully pass messages between them, a path is said to exist between the two localities. A path must exist between two localities before a connection can exist between partners in those localities.

Host Integration Server and SNA Server establish a path using an appropriate method for the network operating system in use. For example, with Microsoft LAN Manager, a named pipe is used; with NetWare, an SPX connection is used. When the two localities are on the same PC, a local pipe is used; this is implemented using shared buffers to increase performance, but is used by the application in exactly the same way as communication with a remote locality.

The DMOD provides communication between dynamic localities and provides guaranteed in-order delivery of messages flowing over paths between localities. If the DMOD loses its path to another locality, it informs the Base.

The following figure illustrates the paths and connections between an SNA server local node and two 3270 emulation programs. Program A has two connections to the local node (one for each of two sessions); program B has one connection to the local node.



LPI Addresses

An LPI address is used to identify each end of a connection. It has three components: locality (L), partner (P), and index (I).

- Locality is a 1-byte identifier that uniquely identifies a locality within a system. This locality corresponds to an SNA server component (local node, link service, 3270 emulator, and so on).
- Partner is a 1-byte identifier that uniquely identifies a partner within the locality. This is not always used, but can be used to distinguish between parts of a component (for example, the 3270 functions in the local node rather than the APPC functions).
- Index is a 2-byte identifier that uniquely identifies a logical entity within the partner. The meaning and use of this field is defined by the communicating partners; it is used to distinguish multiple connections between the same partners (for example, to identify one of many 3270 sessions between the local node and a particular 3270 emulator). The value of zero should not be used as an index value. Applications must assign unique index values for every active LPI connection with the node.

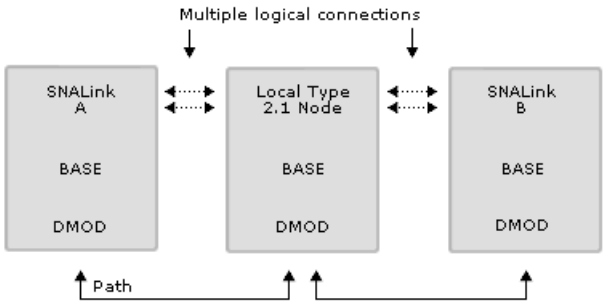
A message flowing over a connection carries a pair of LPIs that identify the source and destination of the message. These are the source LPI and destination LPI of the message; together they identify the connection on which the message is flowing.

Note that more than one connection can exist between any pair of partners. The I values are then used to distinguish the connections. For example, in communications between the local node and a 3270 emulator, the L and P values identify the message as being 3270 data for that local node, and the I value indicates which session the data is intended for.

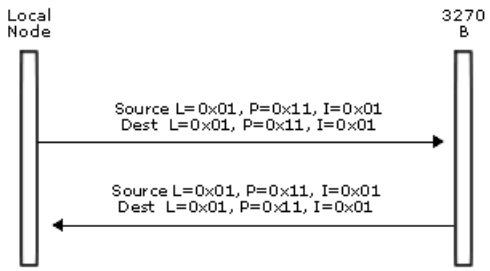
The LPIs are assigned by a combination of the partners and the DMODs when the connection is opened, as described in Making Connections.

Because they are assigned dynamically for each component, the L values are not the same across a whole system. For example, a local 2.1 node locality could be known as locality 4 to one 3270 locality, and locality 6 to a second 3270 locality. However, from the viewpoint of any locality, there exists a unique L value for each remote locality within which a path exists; this L value is used as an index into an internal table that identifies the path to that locality.

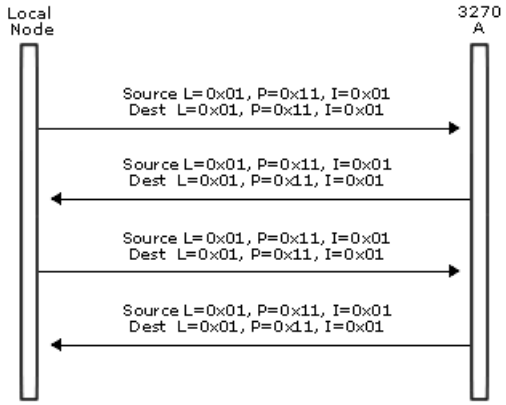
The following figures show an example of the L values that could be used between the components shown in Paths and DMODS, and examples of the LPI values that would be used by the local node on messages flowing between the components. (See Opening the PLU Connection for more information on how the LPI values are assigned and used.)



Example L values



L values specified on messages between the local node and 3270 B



LPI values specified on messages flowing on two different connections between the local node and 3270 A

The Base is called by any piece of code that wishes to send a message. It uses the destination L value on the message to determine where to send it. When the message gets to the remote locality, the Base in that locality routes it to the appropriate partner if the locality contains more than one partner.

Making Connections

Before messages can flow across a connection, the connection must be established, or opened. This is necessary because a partner (P1) does not initially know the LPI address of the partner with which it wishes to communicate; indeed, there may not even be a suitable partner for it to communicate with.

A component of the Base, known as the Resource Locator, and a message with type of Open, known as an Open message, are used to establish a connection between partners.

The following procedure outlines how a connection is established. More specific information is available in The Function Management Interface.

To establish a connection between partners

1. The Open message has two forms: an Open request and an Open response. The Open request contains information on the type of partner P1 is looking for.

   P1 fills in an Open request and calls the Base with it. Since it does not know the LPI address of its partner, it sets the destination LPI values to zero.

2. The Base cannot forward the Open to a particular partner, since it has no destination LPI address. Therefore it passes the Open to the Resource Locator, which attempts to find a locality that will accept the Open. The DMOD has a record of all the localities that could accept this type of Open. The Resource Locator tries each of these localities until the Open is accepted. If no locality is found, the Resource Locator returns a negative response to the Open to inform the sender that no partner could be found.
3. When a remote locality receives an Open, the Base passes the Open to the partner (P2). If P2 can accept the Open, it responds by sending back a positive Open response message to P1.
4. The Open response message returned to P1 contains both the source and destination LPI values for the particular connection. At the end of this exchange, both P1 and P2 know each other's addresses and can communicate over the connection.

Note that the terms "source" and "destination" in the context of LPIs refer to the source and destination of the particular message. Hence, when the 3270 emulator builds a message to send to the local 2.1 node, it needs to swap the source and destination LPIs received on the Open response from the local 2.1 node.

For a detailed example of how LPI addresses are assigned during initialization of the SSCP and PLU sessions, see Opening the PLU Connection.

The DL-BASE/DMOD Interface

This section describes the interface to the Microsoft® Host Integration Server 2000 or the Microsoft SNA Server DL-BASE. It includes a listing of the entry points that an application such as a 3270 emulator can call. These entry points allow messages to be sent to and received from services such as the local 2.1 node.

This section contains:

About DL-BASE/DMOD

The following topics describe an example in which a 3270 emulator is to be adapted to use Microsoft Host Integration Server or Microsoft SNA Server. It needs to communicate with the local 2.1 node.

This section contains:

Initialization

The 3270 emulator should initialize the DL-BASE and then call the dynamic access module (DMOD) to obtain the necessary configuration information. This also registers the user name with the DMOD. It can then obtain further system information such as the Host Integration Server or SNA Server version number, if required.

The functions involved are:

Function	Description
sbpuinit	Initialize the DL-BASE.
sepdcrec	Get configuration information.
sepdgetinfo	Get system information.

The **sbpuinit** entry point should always be called before any other DL-BASE/DMOD entry points except SNAGetVersion. For new emulators, **sepdcrec** should be called after **sbpuinit**. (Because of the order of calls used in older emulators, a call to **sepdcrec** before **sbpuinit** is still supported, but this order is not recommended.)

Sending Messages

The 3270 emulator should insert a message in a buffer and then call the DL-BASE to send it. The message contains source and destination LPIs, which are set up when the connection is opened; see LPI Connections for more information.

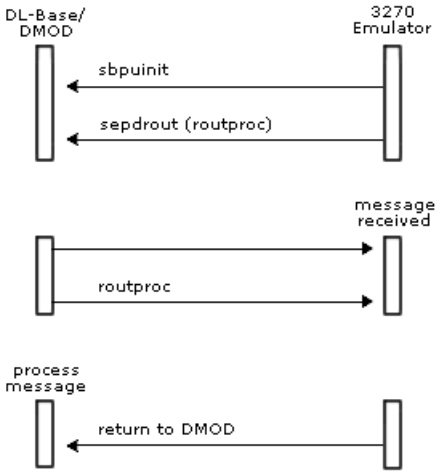
The application can either obtain a new buffer to contain the message to be sent (using sepdbubl), or reuse one in which it previously received a message. The application is responsible for any buffer it has obtained or in which it has received a message; it must either use (or reuse) the buffer to send a message or release it (using sepdburl). If the buffer to be reused does not contain the correct number of elements for the message to be sent, the application can obtain additional elements (using sbpibegt) or release existing ones (using sbpiberl); in this case, it must also ensure that the numelts field in the buffer header indicates the correct number of elements.

The function used to send the message is sbpusend.

Receiving Messages

For Win32 and OS/2

The method for receiving messages from the DMOD is illustrated in the following figure.



After DMOD initialization, the 3270 emulator registers the routing procedure by calling sepdrout. When the DMOD receives a message, it calls the 3270 emulator routing procedure, which can then process the message.

With this approach, there is no context switch between the DMOD thread and the 3270 emulator thread. However, the routing procedure must return control to the DMOD fairly quickly. For instance, it cannot suspend waiting for a keyboard input.

The application must determine whether the received message is for this application or for another application. If the message is not for this application, the routing procedure must return, indicating that the message was not processed. If the application processes the message, it is responsible for freeing the buffer when the processing is finished.

In some cases, the routing procedure can process the message to completion. An alternative is for the routing procedure to put the message on an application queue and then clear an application semaphore. The application can then subsequently process the message.

A further performance gain can be achieved by sending a Status-Resource message (to return credit to the local node, allowing it to send further data) from the routing procedure when a message is received, rather than waiting until the message is processed to completion. This usage is illustrated in Sample Code: Initialization and Routing Procedure. See Pacing and Chunking for more information on credit and flow control.

For Microsoft Windows® version 3.*x*

If an application wishes to schedule its main (non-callback) part on receipt of a message, it can do so by posting a Windows message to its WndProc.

For all operating systems

Note that after the application has received a message it is responsible for the buffer in which the message was received; it must either reuse the buffer to send a message (using sbpusend) or release it (using sepdburl). If the buffer to be reused does not contain the correct number of elements for the message to be sent, the application can obtain additional elements (using sbpibegt) or release existing ones (using sbpiberl); in this case, it must also ensure that the **numelts** field in the buffer header indicates the correct number of elements.

Opening a Connection

Before an LPI connection can be used to transfer data, it needs to be opened. This is performed by sending Open messages, starting with an Open request. The format of the Open messages is defined by the interface being used. For example, the 3270 emulator uses the Function Management Interface (FMI) to communicate with the local 2.1 node.

The interface also defines the initiator of the Open request. In this case, the 3270 emulator sends the Open(SSCP) Request, and the local 2.1 node sends the Open(PLU) Request.

On the Open(SSCP) Request, the 3270 emulator sets all the source and destination LPIs to zero, except for the source index, which can be used by the 3270 emulator for internal routing (for example, to distinguish between two sessions).

The DL-BASE and DMOD ensure that Open messages are routed to a suitable destination. If a routing procedure is used, it should always first call sbpurcvx to process Open responses; when sbpurcvx indicates that it has not processed a received message, and the received message is an Open OK response, the application is informed that the connection was established successfully.

Termination

The 3270 emulator must call sbputerm to free DL-BASE/DMOD resources before it terminates.

MS-DOS-Based 3270 Emulation

This section describes important information for emulator vendors who are writing for the Microsoft MS-DOS® operating system.

For simple emulators that will run as the sole application in a system, much of the following information is unnecessary — the only consideration for vendors of this type of emulator is support for task switching, described in the following topic. However, for emulators that will support background operation it is vital that the facilities provided by the DMOD are used to ensure that this functions correctly.

This section contains:

Task Switching on MS-DOS and Windows 3.x

MS-DOS versions 5 and 6 and Windows 3.x operating systems provide support for task-switching MS-DOS applications. This is a facility that allows the user to switch between multiple applications running on the machine. When an application is not the currently running foreground task, it can be swapped out of memory onto disk by the operating system.

Due to the asynchronous nature of the DL-BASE interface, it is important that this task-switching operation is detected. If the user task switches from an emulator running in the foreground to another application, and the emulator is removed from memory, it is important that the DL-BASE does not attempt to call the emulator's routing procedure with any received messages.

Task-switching support is handled by the DMOD, and need not concern the 3270 emulator writer. However, the DMOD provides a means for the application to make use of the task switch detection. The emulator can call RegisterSwitchProc to register a procedure that will be called whenever an application is being switched out or back in.

If the emulator changes the screen mode (for example, to support model 5 emulation), the task switch notification mechanism must be used to restore the screen mode to its default when the application is being switched out. This is because Windows does not restore the screen mode before attempting to redraw the screen.

Client Environment for MS-DOS-Based Emulators

The MS-DOS-based client Network Access Program (NAP) provides full support for applications that allow switching between foreground and background operation (that is, terminate-and-stay-resident (TSR) applications), including fully preemptive scheduling of a background thread.

To avoid possible interaction problems, it is important that client applications that intend to support background operation make use of the facilities provided by the NAP, rather than implementing separate scheduling mechanisms hooked from MS-DOS interrupts.

This section contains:

Background Operation

During its initialization sequence, the emulator should hook the keyboard interrupt to allow it to scan for the hot key sequence that it uses to switch between foreground and background operation.

When the emulator has completed its initialization and wishes to go resident, it must call CMDGoTSR passing the address of a procedure that is used to start a background thread of execution. The call executes an MS-DOS TSR call to force the emulator to be resident, and will never return control to the emulator.

The background thread can be used to perform any work items that are required (such as processing messages received from the local node, initiating a file transfer, and so on), but the emulator must not write any output to the screen while in background mode.

When the emulator detects a hot key sequence that commands it to return to the foreground, it must first call CMDStopFG to suspend the current foreground thread and gain control of the screen area. If this call returns a nonzero value, the foreground thread has been stopped within MS-DOS. In this case, the emulator must not switch into the foreground, and should restart the foreground thread.

Scheduler

The scheduler provided by the NAP component takes complete control of scheduling all applications running in the system. It is fully preemptive, and allows several processes to run concurrently. Only the current foreground process is permitted to write to the screen.

The entry point provided by the 3270 emulator on the CMDGoTSR call is executed as a fully independent background thread of execution. Scheduling between the foreground application's thread and the emulator background thread occurs in response to:

- MS-DOS interrupts.
- Timer ticks.
- API calls that affect semaphores.

Semaphores

In any preemptively scheduled environment, the use of semaphores is vital to protect data structures that can be accessed by more than one thread of execution. The standard semaphore functions (Set, Clear, Wait, Request) are provided as DMOD entry points.

For example, if the emulator uses the routing procedure method of receiving messages, the routing procedure will be called from a different thread context to the main application thread. This may require the use of semaphore protection around emulator data structures.

DL-BASE/DMOD Entry Point Summary

The following table shows entry points divided into the categories DL-BASE, DMOD, and buffer management, and listed in alphabetic order within each category.

DL-BASE entry points

DL-BASE Entry points	Description
sbpuinit	Initialize the DL-BASE.
sbpurcvx	Process Opens from routing procedure.
sbpusend	Send message.
sbputerm	Terminate.

DMOD entry points

DMOD entry points	Description
routproc	Sample routing procedure.
sepdchnk	Get FMI chunk size.
sepdcrec	Get user and diagnostics records from configuration file.
sepdgetinfo	Get SNA server system information.
sepdrout	Set up routing procedure (OS/2, Windows 2000, Windows NT®, Windows 98, Windows 95, and Microsoft MS-DOS only).
sepwrout	Set up routing procedure (Microsoft Windows 3.*x* only).
(for MS-DOS)	
CMDGoTSR	Start background thread and go resident.
CMDSemClear	Standard semaphore functions for MS-DOS environment.
CMDSemRequest	
CMDSemSet	
CMDSemWait	
CMDStartFG	Resume scheduling of current foreground process.
CMDStopFG	Suspend current foreground process.
RegisterSwitchProc	Register a task-switch detection procedure.

Buffer management entry points

Buffer management entry points	Description
sbpibegt	Get buffer element.
sbpiberl	Release buffer element.
sepdbubl	Get buffer.
sepdburl	Release buffer.

Note The PASCAL calling convention is used for all entry points including routing procedures on MS-DOS, Windows 3.x, and OS/2. The standard-call convention (CDECL) is used on Windows 2000, Windows NT, Windows 98, and Windows 95.

Note The format of buffer headers and elements is described in Messages. The formats of individual messages contained in buffers are defined in FMI Message Formats.

Sample Code: Initialization and Routing Procedure

This section contains an outline of source code for receiving messages from the DMOD.

Note that TRACE*n*() is a macro used to specify data to be traced; this data can include variable parameters. The value *n* identifies the severity level of the trace. The unmatched parentheses are deliberate; they are resolved by the expansion of the macro.

```
/*****************************************************************/
/* Sample code for initialization and routing procedure.         */
/*****************************************************************/

HSEM  dummysem = NULL;      /* This semaphore is never used      */

/*****************************************************************/
/* Initialization procedure                                      */
/*****************************************************************/
USHORT init_proc()
{
  COM_ENTRY("initp");
  rc = sbpuinit(&dmodsem, CLIENT, CES3270, username);
  TRACE4()"DMOD initialized, rc=%d",rc));
  if (rc == NO_ERROR)
  {

    /*****************************************************************/
/* The procedure routproc will be called whenever a message is   */
/* received by the DMOD. This is used to post back the application, */
/* but take care to protect any queues against concurrent access by */
/* multiple threads.                                             */
/*****************************************************************/
    rc = sepdrout(routproc);
    TRACE4()"Rout proc set up, rc=%d",rc));
    if (rc == NO_ERROR)
    {
      /* Other initialization here                               */
    }
  }
  return (rc);
}
/*****************************************************************/
/* The routine routproc is called whenever the DMOD receives a   */
/* message or a status indication                                */
/*****************************************************************/
USHORT FAR _loadds routproc(buf, srcl, status)
BUFHDR FAR *buf;        /* Buffer that has been received         */
USHORT      srcl;       /* Locality from which buffer was received  */
USHORT      status;     /* Reason for call                       */
                        /* CEDINMSG = message received           */
                        /* CEDINLLN = path error occurred (on srcl) */
{
  COM_ENTRY("routp");   /* initialize rc=FALSE                   */

                        /* Call the DL BASE to handle re-resource   */
                        /* location                              */
  if (!sbpurcvx(&buf, srcl, status))
  {
    switch (status) {
      case CEDINMSG:
        if (buf->destp == S3PROD)         /* Is the message for us? */
        {
          /*****************************************************************/
     /* Process the received message.                              */
     /*                                                            */
     /* If the message is DATAFMI on the PLU-SLU session, and the  */
     /* application has requested to use flow control on the       */
     /* session, then this processing should include:              */
     /*                                                            */
```

```
      /* -  increment number of messages received by the client      */
      /* -  check whether the number received exceeds the threshold  */
      /*    for normally returning credit to the node. If so, check  */
      /*    whether it's OK to return credit (e.g. not short of      */
      /*    buffers), and if OK send a status-resource message to    */
      /*    the node to give it credit to send more messages to the  */
      /*    client.                                                  */
  /************************************************************************/
           rc = TRUE;
           TRACE2()"Routing proc got message at %p",buf));
         }
         else
         {
           TRACE2()"Routing proc did not take message at %p",buf));
         }
         break;

       case CEDINLLN:
         TRACE2()"Path error on %d",srcl));
             /*****************************************************************/
         /* Process the path error status.                           */
  /************************************************************************/
         break;
     }

     /*********************************************************************/
     /* If the message/status cannot be completely processed here,   */
     /* the app can queue the message and clear a semaphore for the  */
     /* main thread to continue the processing.                      */
  /************************************************************************/
   } else {
     rc = TRUE;       /* DLBase handled the message on our behalf    */
   }
   /* Returning a value of TRUE indicates that we processed the      */
   /* event return(rc);
  }
```

The Function Management Interface

The Function Management Interface (FMI) provides applications with direct access to SNA data flows and information about SNA control flows by means of status messages. This section provides information about the SNA sessions and connections over which FMI messages can flow, and summarizes the messages. The FMI is particularly suited to the requirements of 3270 emulation applications.

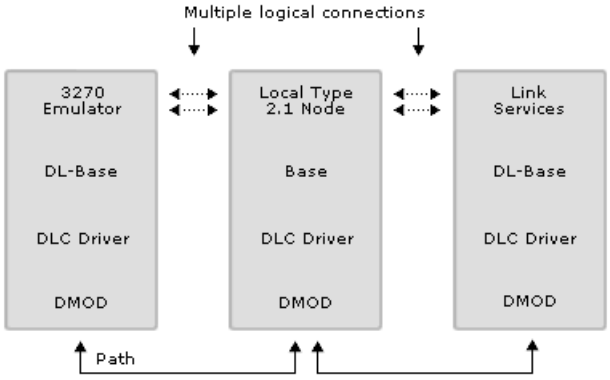
This section contains:

FMI Concepts

The local node provides the SNA layers of path control, transmission control, and data flow control (DFC), as well as logical unit (LU) services — see the following figure. In terms of the SNA layers, the FMI is between presentation services and DFC. This means that most of the SNA protocol handling is performed by the local node. In particular, the local node's DFC layer is responsible for the state changes associated with chaining, bracket, and quiesce protocols.

The FMI is defined in terms of the messages that are sent across the interface. Note that this is logically distinct from the definition of the DL-BASE/DMOD interface, which defines the mechanism for sending messages between two components in Microsoft® Host Integration Server or Microsoft SNA Server (for example, between the local node and the 3270 emulator).

The FMI is used by LU types 0, 1, 2, and 3, but not by LU type 6.2. It provides access to the SSCP-LU session as well as the main PLU-SLU session (see Sessions and Connections for more information on these sessions). An application can use the FMI to access multiple sessions and hence multiple LUs, simultaneously.



In this example, the 3270 emulator on the client communicates over the LAN with the local node on the server machine by exchanging messages. The content and format of the messages are defined by the FMI. The DMOD component is used to transport the messages, but does not interpret them. The local node provides the SNA service for formatting the message. The link service and the data link control (DLC) driver are responsible for transferring data between the local node and the DLC adapter.

This section contains:

- Sessions and Connections
- Application Flags

Sessions and Connections

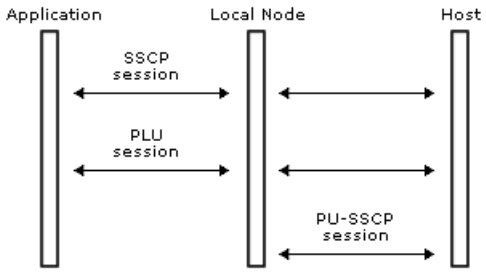
An application using the FMI can communicate with the host on two SNA sessions:

- The SSCP session, between an SNA server LU and the host system services control point (SSCP), provides information on the activation of the LU and supports communication with the SSCP for commands such as character-coded and field-formatted logon and logoff commands. There is one SSCP session for each SNA server LU.
- The PLU session, between an SNA server LU and the host primary logical unit (PLU), is the main session for data transfer between the local application and the host application. There is one PLU session for each SNA server LU.

The local node communicates directly with the host on the PU-SSCP session.

- The PU-SSCP session, between the physical unit (local node) and the host SSCP, supports the reporting of alert information and link statistics to the host SSCP.

The three sessions are illustrated in the following figure:

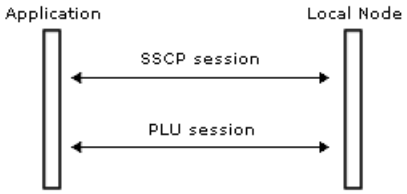


The application can communicate with the local node by means of two LPI connections. Rather than specifying the session on which a message is to flow, the application sends the message to the local node on one of these connections; the local node then routes it to the appropriate SNA session.

The connections are used as follows:

- The SSCP connection is used for the initial start-up and logon information for a 3270 session. The Host Integration Server or SNA Server 3270 emulation programs also send network management information, such as user alerts and Response Time Monitor (RTM) statistics, to the local node on this session. See The SSCP Connection for more information on this connection.
- The PLU connection is used for the transfer of application data, and for status and flow control messages between the application and the local node. See The PLU Connection for more information on this connection.

The connections are illustrated in the following figure.



These connections are specific to the local node and the application. Data and status messages passed across a connection result in SNA data and SNA control requests being sent on the appropriate SNA session. Similarly, SNA data and SNA control responses received on an SNA session result in data and control messages being passed to the application on the appropriate connection.

The relationship between the three SNA sessions and the two connections is as follows:

- SNA messages on the SSCP session from the host to the local node result in messages from the local node to the application on the SSCP connection. Messages from the application to the local node on the SSCP connection normally result in SNA messages on the SSCP session from the local node to the host (with the exception of network management information, which results in messages on the PU-SSCP session — see below).
- SNA messages on the PLU session from the host to the local node result in messages from the local node to the application on the PLU connection. Messages from the application to the local node on the PLU connection result in SNA messages on the PLU session from the local node to the host.
- SNA messages on the PU-SSCP session from the local node to the host are generated by messages from the application to

the local node on the SSCP connection. When the application sends network management information such as 3270 user alerts on the SSCP connection, the local node distinguishes it from other data on this connection (which normally corresponds to the SSCP session) and sends the appropriate information on the PU-SSCP session to the host. See 3270 User Alerts for more information.

Note the distinction between these SNA sessions and 3270 emulation sessions. A 3270 emulator can have more than one 3270 emulation session; for each emulation session, there are separate SSCP and PLU sessions.

Each connection between the application and the local node is opened, managed, and closed separately. This means that an application must maintain a separate internal control block containing the LPI pair, message keys, and state of the connection for each of the SNA sessions associated with each 3270 emulation session; for example, an application using three 3270 emulation sessions, each with an SSCP session and a PLU session, will require six control blocks.

An application identifies the connection (and hence the session) to which a particular message belongs using the LPI pair present in the message. On a received message, the destination index (I) value contains the application's identifier for the connection, and the source I value contains the local node's identifier for the connection. These are reversed for messages sent by the application.

The application selects the LU within the local node that it can use for communications by the relationship in the configuration table between the LU and APPL records (see Opening the SSCP Connection). The application may be unaware of which LU it accesses if the LUs are arranged within LU groups.

Application Flags

Application flags are included on the following messages:

- All Data messages (both inbound and outbound)
- Status-Acknowledge(Ack) (outbound only)
- Status-Acknowledge(Nack-1) (outbound only)
- All Status-Controlmessages (both inbound and outbound)

These flags represent key indicators of the state of the session to which the message relates and are closely related (but not always equivalent) to the request header or response header (RH) indicators in the SNA request or response. Note that for inbound messages, applications need to set the flags on **Data** messages and **Status-Control** messages only.

For outbound messages, the local node sets the application flags to reflect the contents of the RH in the corresponding SNA message. The local node performs checks on the SNA message before sending it to the application; therefore, the application can assume that the RH indicators obey the SNA protocols and need not perform its own checks. The application's task in interpreting the application flags is much simpler than if the local node presented the message with the uninterpreted RH. For example:

- If the application specified the segment delivery option when the PLU connection was opened (see Opening the PLU Connection), the end chain indicator (ECI) on an SNA request will occur on the first segment of the last request unit (RU) in a chain, but the chain is not complete until the last segment of that RU is received. In this case, the local node manipulates the application flags so that the ECI flag is set in the last segment rather than the first.
- Applications using TS profile 4 (transmission service profile 4) on the PLU session can receive the DR2 (definite response 2) RH indicator in combination with DR1 (definite response 1) or ER (exception response) to give RQD2, RQD3, RQE2, and RQE3 requests. The local node interprets the RH indicators and sets the COMMIT application flag accordingly.

For inbound Data and Status-Control messages, the application should set the flags to control session characteristics such as chaining, direction control, and brackets. For **Status-Acknowledge** messages, the local node generates an SNA response and sets the RH indicators using information saved from the corresponding request; the application does not need to set the flags on this message.

For information on application flag usage when FMI chunking is being used, see Chunking.

In most cases, the application will not need to consider the application flags on Status-Acknowledge(Ack) messages, which derive from the RH indicators on the corresponding response. However, certain applications do require access to the RH flags on responses — for example, transaction-processing applications using TS profile 4 can receive the DR2 flag on responses, which will appear as the COMMIT flag in the application flags.

The application flag usage on Status-Control messages is derived from the RH indicators in the corresponding DFC or session control (SC) RU. Applications may need to be aware of the RH flags for Status-Control messages. For example, LUSTAT request type 6 is a "no-op" used solely to allow RH flags to be sent when no other request is allowed. The local node delivers the request to the application as a Status-Control(LUSTAT) Request with the relevant application flags set. See *SNA Format and Protocol Reference Manual: Architectural Logic* (IBM publication SC30-3112) for summaries of valid RH usage for DFC RUs and of valid RH indicators for SC requests.

In the summary of the application flags below, bits are numbered with bit 0 as the most significant bit in a byte and bit 7 as the least significant. An application flag is set if the relevant bit for the flag is 1 and not set if the bit is 0.

Flag 1 occurs in all messages. The meanings of the individual bits and values that can be ORed together to set them are listed below.

Bits in flag 1	**Meaning**
FMHI [bit 0, flag 1] Value: AF_FMH (0x80)	Function management header indicator — set if a function management header is present in the message, or if the message is a function management data network services (FMD NS) request. Only valid on Data messages. This flag is always set for 3270 user alerts, which are sent on the SSCP connection (see 3270 User Alerts).

BCI [bit 1, flag 1] Value: AF_BC (0x40)	Begin chain indicator — set if this message starts a chain (see Outbound Chaining and Inbound Chaining).
ECI [bit 2, flag 1] Value: AF_EC (0x20)	End chain indicator — set if this message ends a chain (see Outbound Chaining and Inbound Chaining).
COMMIT [bit 3, flag 1] Value: AF_COMM (0x10)	Commit indicator — set if chain carries DR2 (definite response 2).
BBI [bit 4, flag 1] Value: AF_BB (0x08)	Begin bracket indicator — set if chain carries BB (begin bracket). Note that this does not necessarily indicate that the bracket has been initiated (see Brackets).
EBI [bit 5, flag 1] Value: AF_EB (0x04)	End bracket indicator — set if chain carries EB (end bracket). Note that this does not indicate that the bracket has terminated (see Brackets).
CDI [bit 6, flag 1] Value: AF_CD (0x02)	Change direction indicator — set if chain carries CD (change direction); see Direction.
SDI [bit 7, flag 1] Value: AF_SD (0x01)	System detected error indicator — set if the local node detects an error in outbound data; see Outbound Data.

Flag 2 occurs in all messages except **Status-Control(STSN)**, where the indicators included in this byte are not applicable. The meanings of the individual bits and values that can be **OR**ed together to set them are listed below.

Bits in flag 2	**Meaning**
CODE [bit 0, flag 2] Value: AF_CODE (0x80)	Alternate code indicator — set if the alternate code set (usually ASCII) is used for this Data message. Note that function management headers are unaffected by the code selection indicator.
ENCRYP [bit 1, flag 2] Value: AF_ENCR (0x40)	Enciphered data indicator — set to indicate that the information in the **Data** message is enciphered under session level cryptography protocols. Users should note that they must provide the necessary support for data encryption; the Host Integration Server or SNA Server local node does not support cryptography.

ENPAD [bit 2, flag 2] Value: AF_ENPD (0x20)	Padded data indicator — set in conjunction with the ENCRYP flag to indicate that the data was padded at the end to the next integral multiple of eight bytes before encipherment.
QRI [bit 3, flag 2] Value: AF_QRI (0x10)	Queued response indicator — set if the response to this request is to be queued in the transmission control and data flow control layers. This flag is only significant for inbound messages.
CEI [bit 4, flag 2] Value: AF_CEI (0x08)	Chain ending indicator — set on a message corresponding to an outbound SNA request with EC (end chain) and BBIU (begin basic information unit). This flag is provided solely for the use of SNA server components; your application should not attempt to use it.
BBIUI [bit 5, flag 2] Value: AF_BBIU (0x04)	Begin basic information unit indicator — set on a message corresponding to an outbound SNA request with BBIU (begin basic information unit). This flag is provided for the use of SNA server components and for applications using segment delivery and outbound pacing together (see Pacing and Chunking); your application should not attempt to use it.
EBIUI [bit 6, flag 2] Value: AF_EBIU (0x02)	End basic information unit indicator — set on a message corresponding to an outbound SNA request with EBIU (end basic information unit). This flag is provided solely for the use of SNA server components; your application should not attempt to use it.
RBI [bit 7, flag 2] Value: AF_RBI (0x01)	Real BID indicator — set on **Status-Control(BID) Request** messages from the local node only. 0x01 indicates that the message is due to an SNA BID RU, 0x00 indicates that the message is due to an outbound FMD (function management data) RU with BB (begin bracket) set.

The SSCP Connection

The application's SSCP connection to the local node provides access to the SSCP session between the Microsoft Host Integration Server or Microsoft SNA Server secondary LU and the host SSCP.

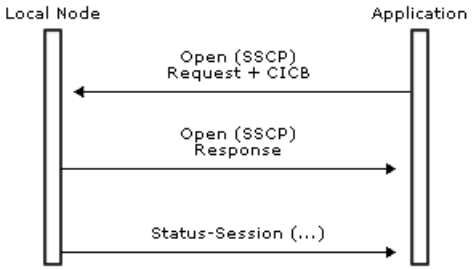
For simplicity, this section describes the SSCP connection as if an application only uses a single SNA server LU (and therefore a single SSCP connection); in practice, applications can use multiple LUs.

This section contains:

Opening the SSCP Connection

An application gains access to the SSCP session by opening an SSCP connection to the local node. To do this an application sends an Open(SSCP) Request message to the local node, which responds with an Open(SSCP) Response. The local node follows a positive **Open(SSCP) Response** with a Status-Session message reporting the current state of the SSCP session (see Using the SSCP Session).

The message flow is shown in the following figure. For a more detailed message flow diagram, including the LPI values used during initialization of both the SSCP and PLU sessions, see Opening the PLU Connection.



This section contains:

- LU Groups
- Resource Location for Open SSCP

LU Groups

Note that Host Integration Server and SNA Server supports LU groups; a group consists of a number of LUs of the same type (3270 display LUs or LUA LUs) such that any of the LUs in the group can be used for the same task.

If an application sends an Open(SSCP) Request specifying the name of a 3270 display LU group, the local node can select any LU within the group to be used by the application. LUA LU groups are used in the same way, except that the application can specify either the name of the group or the name of any LU within the group to access the group.

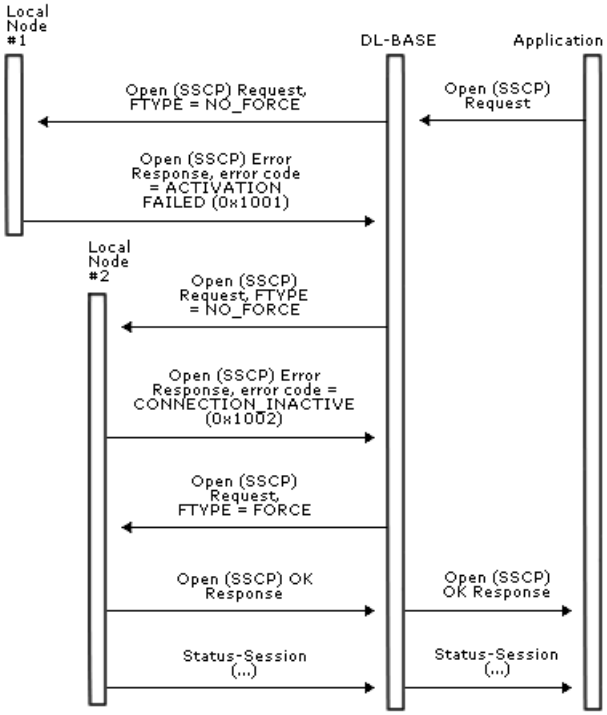
Resource Location for Open SSCP

When attempting to find a free LU across more than one local node, the application does not need to be aware of which local node owns the LU. The DL-BASE is responsible for finding a suitable local node, using the mechanism described below. This mechanism need not concern the application; the description is intended to assist in interpreting traces of the message flows involved.

The open force type field in the Open(SSCP) Request specifies either a forced or nonforced Open. If the LU for which the Open is intended does not have an active SSCP session because its link is inactive, a forced Open instructs the local node to attempt to activate the link and the SSCP session; a nonforced Open succeeds only if the SSCP session is already active, and otherwise returns with an error code indicating the state of the LU's connection.

When the application issues the Open(SSCP) Request, it does not set the open force type field. The DL-BASE then issues a nonforced Open to each node in turn until it finds an LU that already has an active SSCP session. If none of these Opens succeeds, the DL-BASE issues a forced Open to the node that returned the "best" error code — that is, the one most likely to be able to activate the session.

The sample message flows in the following figure illustrate this process for two local nodes. The DL-BASE tries each in turn, using nonforced Opens; the error code from node #2 indicates that it is more likely to be able to activate the SSCP session than node #1, so the DL-BASE sends a forced Open to node #2. The application is aware only of the first request and its response.



To allow applications to restart after a disastrous failure (such as terminating the 3270 emulation program), the local node will also accept an Open(SSCP) Request from an application that has failed and restarts (providing the same source LPI fields are used). In this case a TERM-SELF message is sent to the host if the LU is bound.

The SNA server LU through which the application communicates is selected by the relationship between the APPL record and the LU or LU group record in the configuration file. The application specifies its name using the source name field on the Open(SSCP) Request; the local node fills in the LU or LU group number, selects an unused LU within the LU group (if the association is to an LU group), and informs the application of this LU number on the Open(SSCP) OK Response.

The Open(SSCP) Request specifies:

- The source application name.
- A resource identifier that can be used by the application to correlate the Open(PLU) Request that is sent to the application (see Opening the PLU Connection).
- A connection information control block (CICB) that specifies the RH usage checks that the local node should perform for the LU. If the field for a code is set to 0x01, then that receive check will be carried out by the DFC layer of the local node on data arriving from the host. The corresponding send checks are unaffected and are always performed. The CICB is provided because these receive checks are optional in SNA; however, it is anticipated that most applications will require all these checks to be performed (all values set to 0x01).

- An indicator that specifies whether the application is to be treated as high or low priority. All SNA server 3270 LUs are marked as high priority (printers do not send significant data inbound). The effect of high priority is to allow data to be progressed faster to the host when the link is busy.
- An indicator that specifies whether the application is an LUA application. This determines whether the local node and the application will communicate using the LUA variant of the FMI (see FMI Concepts).
- An indicator that specifies a nonforced or forced Open. This determines whether the local node will attempt to activate the SSCP session if it is not currently active.

The Open(SSCP) Request can fail for one of several reasons, which can be determined from the error codes on the Open(SSCP) Response sent to the application:

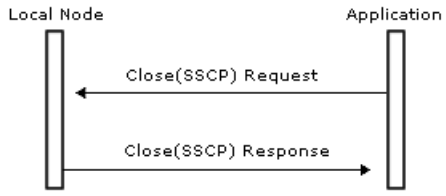
- The local node may still be initializing (retrieving information from the configuration file). In this case, the application can retry immediately.
- The configuration file may not have an entry for the application, or the application record in the configuration file may not point to an LU or LU group record.
- For a nonforced Open, the SSCP session may be inactive.

Closing the SSCP Connection

To close the SSCP connection, an application sends a Close(SSCP) Request to the local node, which responds with a Close(SSCP) Response. The **Close(SSCP) Request** is unconditional; the **Close(SSCP) Response** always reports that the connection was successfully closed. The **Close(SSCP) Response** is provided so that applications can determine when outstanding data and status messages on the session have been delivered.

If the LU is bound, the local node sends a TERM-SELF message to the SSCP on behalf of the application to elicit an UNBIND. An application that merely wishes to be unbound need only issue Close(PLU) (see Closing the PLU Connection). Normally the SSCP connection can be maintained while the application task is active, even if it is idle.

Closing the connection invalidates the LPI pair for the connection but does not alter the state of the SSCP session. The message flow is shown in the following figure.



Using the SSCP Session

When the application has opened an SSCP connection, it has access to the SSCP session and can send data to the host SSCP.

This section contains:

SSCP Session Characteristics

For SNA type 2.1 nodes, the SSCP session uses FM (function management) profile 0 and TS (transmission service) profile 1. This combination of profiles gives the following session characteristics:

- The primary and secondary half-sessions both use immediate request mode.
- The primary and secondary half-sessions both use immediate response mode.
- Only definite-response single RU chains are allowed.
- The maximum RU size is limited to 256 bytes.
- DFC RUs are not supported.
- Pacing is not supported.
- Identifiers are used (rather than sequence numbers) on the normal flows.

This implies that the SSCP connection has the following characteristics:

- All Data messages have the ACKRQD (acknowledgment required) field set.
- All Data messages have the BCI (begin chain indicator) and ECI (end chain indicator) application flags set.
- Status-Control messages do not flow on the connection.
- Status-Session messages from the local node to the application only report changes in the activation state of the session.
- The chaining, bracket, confirmation, and recovery protocols (described in The PLU Connection) do not apply.

Using the SSCP connection, the application can send and receive **Data** messages corresponding to FMD NS (function management data network services) (session services) requests and FMD data requests. Examples of FMD NS (session services) requests are:

- INIT-SELF: Requests from the secondary to the host SSCP requesting that the SSCP assist in the initiation of a session to the host PLU, effectively requesting a BIND (see Opening the PLU Connection).
- TERM-SELF: Requests from the secondary to the host SSCP requesting that the PLU-SLU session be terminated with an UNBIND (see Closing the PLU Connection).
- Character-coded requests: Requests such as logon, logoff, or test commands from the secondary display, or a logon prompt from the host application.
- NOTIFY: Requests used by the secondary to notify the host SSCP that a device is available after a BIND was rejected with sense code 0x0845; for example, where a device emulator supports logical power-off.

The local node sends a NOTIFY request to the SSCP on behalf of the LU whenever the application's SSCP connection state changes while the LU is active. A NOTIFY (vector key 0x0C with byte 5 set to 0x03), which can act as secondary LU, is sent in the following cases:

- When an Open(SSCP) Request is received when the LU is already active.
- When an ACTLU request is accepted when the SSCP connection is already opened.

A NOTIFY (vector key 0x0C with byte 5 set to 0x01), which cannot currently act as secondary LU, is sent in the following cases:

- When an ACTLU is received when the SSCP connection is not open.
- When a Close(SSCP) Request is received when the PLU session is not bound.
- When an UNBIND request is received when the SSCP connection is not open.
- In addition, the long response including the NOTIFY vector is used to ACTLU requests.

These NOTIFY messages can be used by the host in conjunction with the negative response 0x0845 that the local node gives to a BIND received when the SSCP connection is not open (see Opening the PLU Connection).

SSCP Session Status

While the SSCP connection is open, the local node reports the initial state and any subsequent changes of state of the SSCP session to the application using Status-Session messages. There are four distinct **Status-Session** status codes that can occur for the SSCP connection:

- No-Session: The SSCP session between the SNA server LU and the host SSCP is not active because the SNA server PU and/or LU is not activated. The **Status-Session** carries a qualifying status code to indicate why the SSCP session is inactive. The application cannot use the SSCP connection to send data to the host SSCP. The qualifiers are:

  PU-INACTIVE: ACTPU has not been received or DACTPU has been received.

  PU-ACTIVE: ACTPU(COLD) has been received from the SSCP.

  PU-REACTIVATED: ACTPU(COLD) has been received while the PU was active (the application is not informed if ACTPU(ERP) is received while the PU is active).

  LU-INACTIVE: ACTLU has not been received, or DACTLU has been received.

- Link-Error: The SSCP session between the SNA server LU and the host SSCP is not active, due to a DLC link error. The **Status-Session** carries a qualifying status code that gives the error code reported by DLC. The application cannot use the SSCP connection to send data to the host SSCP.

  Note that this session state is reported when the local node is informed that the locality containing the Host Integration Server or SNA Server SDLC link service has been lost due to a path failure; the qualifier 0x0D is used. The link service will close the link when it is informed of the path error so the application can treat this as an outage.

- LU-Active: The SSCP session is active due to the receipt of ACTLU. The application can use the SSCP connection to send data to the host SSCP.
- LU-Reactivated: The SSCP session has been reactivated due to the receipt of an ACTLU from the host SSCP. The SSCP connection is still active, but data may have been lost.

Status-Session Codes describes the **Status-Session** status codes.

RTM Parameters

The Status-RTM message is sent to the application by the local node to indicate the Response Time Monitor (RTM) parameters being used by the host. The host can specify the following parameters:

- Whether RTM measurement is active or inactive.
- Whether local display of RTM data by the application is permitted.
- The definition by which response times are to be measured:

  Until the first character of a response is written to the screen.

  Until the keyboard is unlocked.

  Until the application is allowed to send data (CD or EB received).

- The boundaries by which response times are to be classified into time bands.
- The initial values of the counters, which indicate how many responses have been received in each time band (as defined by the boundaries).

The local node is responsible for interpreting the response times reported to it by the application, and for sending RTM statistics to the host when required; the application is responsible for measuring the response times and reporting them to the local node. (The application reports response times to the local node using the Status-Acknowledge message — see Response Time Monitor Data for more information on measuring and reporting response times).

If the application does not wish to provide a local display of RTM data, it only needs to determine whether RTM measurement is active and, if so, the definition by which response times are measured; it can ignore the other parameters. If RTM measurement is not active, the application need not measure and report response times.

If the application wishes to provide a local display of RTM data, it should use the information from the Status-RTM message to ensure that the local interpretation of response times matches the interpretation used by the host. In particular, it should not display RTM data at all if the Status-RTM message indicates that local display is not permitted (or if the "permission to view RTM data" field in the 3270 user configuration record indicates that it is not permitted). The application is responsible for maintaining its own RTM statistics for local display; that is, for classifying the response times according to the boundaries specified by the host and maintaining counts of responses in each category. Although the local node maintains these statistics for sending to the host when required, it does not report them to the application.

> RTM statistics are maintained for a specific LU, not for a specific application's use of that LU. This means that when the **Status-RTM** message is received at start of day the counters can be nonzero to include a previous use of the LU. The counters are only reset when the host requests the local node to reset them or when the local node sends unsolicited RTM data to the host.

3270 User Alerts

A Host Integration Server or SNA Server 3270 emulation program can send 3270 user alerts to the local node on the SSCP connection. This allows the local node to route each alert to the appropriate host for the 3270 session on which it was sent.

To send a 3270 user alert, the application should send it as a Data message on the SSCP connection. The local node will recognize it as a 3270 user alert if both of the following are true:

- The FMHI (function management header indicator) bit in the application flag 1 byte is set.
- The first three bytes of the data are 0x41038D, indicating an NMVT (Network Management Vector Transport).

The local node sends the alert to the appropriate host for the 3270 session on which it was received. If a relative time subvector is present (0x42) with increment type 0xEF (sequence), then the local node will set the sequence number in each message (starting at 1 from power-up and incrementing by 1 for each message sent). Comm Server allows sequence number values up to 2^16. Apart from this, the local node does not alter the contents of the alert.

Note that there can be some delay before the application receives a response to the alert; the response is sent on the SSCP connection in the same way as other data on this connection. The application must not send further data on the SSCP connection (including further alerts) until it has received a response to this alert.

The PLU Connection

The application's PLU connection to the local node provides access to the PLU session between the Microsoft Host Integration Server or Microsoft SNA Server LU and a PLU (primary LU) in the host.

This section describes:

- How an application opens and closes its PLU connection.
- The use of the PLU connection.

For simplicity, this section describes the PLU connection as if an application uses only a single SNA server LU (and therefore a single PLU connection); in practice, applications can use multiple LUs.

This section contains:

Opening the PLU Connection

The opening of the PLU connection is closely associated with the establishment of the PLU session. The local node opens the PLU connection when it receives a BIND command from the host for an LU for which an application has previously opened an SSCP connection. Possible sequences are:

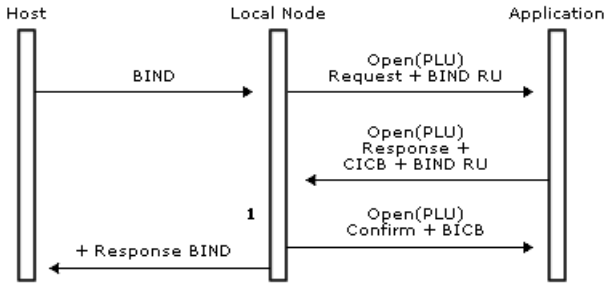
- An application opens its SSCP connection and sends a character-coded logon request or INIT-SELF request to the host SSCP. A host PLU subsequently sends a BIND request to the SNA server LU, and the local node opens the PLU connection.
- A host PLU sends an unsolicited BIND command to the SNA server LU. If the SSCP connection for the LU is open, the local node opens the PLU connection. If the local node is supporting NOTIFY, then the host can be configured to send a BIND when it receives the NOTIFY message sent by the local node when the application opens its SSCP connection (see The SSCP Connection).
- A host PLU sends a BIND command to the SNA server LU. If the SSCP connection for the LU is not open, the local node returns a negative response to the BIND request. The sense code used is 0x0845 (NOTIFY will be sent). The local node does not open the PLU connection. In this case the local node sends NOTIFY when the SSCP connection is opened (see The SSCP Connection).

To successfully open the PLU connection, the local node sends an Open(PLU) Request to the application. The application responds with an Open(PLU) OK Response. Finally the local node sends an Open(PLU) OK Confirm to the application. This exchange of messages opens the PLU connection and establishes the PLU session. It should be noted that a successful PLU opening sequence is a "three-way handshake," in comparison to the opening of the SSCP connection, which is a "two-way handshake."

The Open(PLU) Request is delivered to the application using the SSCP connection for the LU. The Open(PLU) Request contains the application name and open resource identifier to allow applications to correlate the PLU and SSCP connections.

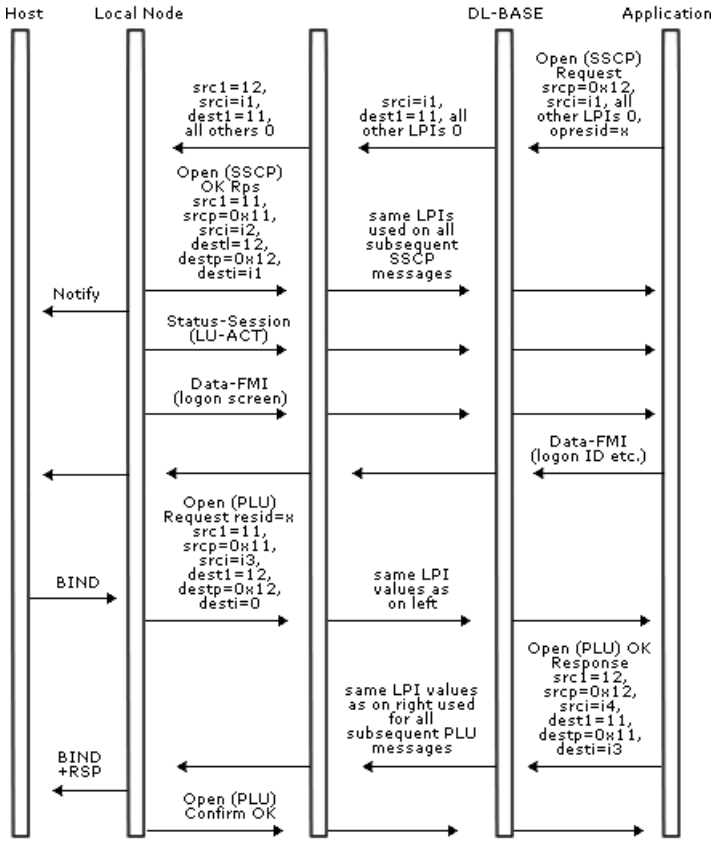
The Open(PLU) Request indicates the logical unit that the BIND request was directed to, references the resource identifier supplied in the Open(SSCP) Request for that LU, and carries the actual BIND RU received from the host (see Open(PLU)). It also carries the maximum RU sizes, chunk sizes (if appropriate), and pacing windows for the PLU session, to enable the application to determine the initial credit if it wishes to be involved in outbound pacing (see Pacing and Chunking).

The message flow for a successful opening of the PLU connection (on receipt of a nonnegotiable BIND) is shown in the following figure. Note that the BIND parameters are verified (at [1]) only when the application has supplied the BIND check table index as part of the CICB.



The following figure shows the message sequence for the initiation of both the SSCP and PLU sessions, including details of where the LPI values are assigned. (The application's source P value of 0x12 indicates that it is a 3270 emulator; see Open(SSCP) Request for more details of how the source LPI values are set.) The message flow shown assumes that the connection to the host is already established and that both the configuration and the BIND are valid.

After this message sequence, there are two valid sets of LPI values, one for the SSCP session and one for the PLU session. The application can access either session at any time until UNBIND and can use the LPI values to distinguish between received data on the two sessions.



This section contains:

- BIND Checking

BIND Checking

The Open(PLU) OK Response contains the connection information control block (CICB), which allows the application to tailor certain characteristics of the connection and contains information used in BIND verification. Note that the local node verifies the BIND parameters carried on the **Open(PLU) OK Response**; it does not maintain a copy of the original BIND RU from the host. If the BIND is negotiable, the application is permitted to modify the parameters in the BIND RU, but if it is nonnegotiable the application should return the BIND RU unmodified. A negotiable BIND flag is provided in the Open(PLU) Request.

While many characteristics of the PLU session are determined by the BIND parameters, the application can select certain characteristics by specifying fields in the CICB; see the following table. More detailed information on CICB usage and the effect on the PLU session of selecting various CICB options is given in context in the topics of this section that deal with PLU session characteristics such as chaining and pacing.

The BIND is verified using a BIND check table entry (whose index is specified in the CICB); the entries in this correspond to the various fields in the BIND. The BIND check table entries are stored in the configuration file. For example, the BIND check table entry can specify that the BIND be accepted if the secondary chain response protocol is either "definite response" or "definite or exception response" (byte 5 bits 2 and 3 = B10 or B11); this would be appropriate if the application did not want to send RQE chains.

Connection Information Control Block Usage

Field	Explanation
Segment delivery option	A value of 0x00 indicates that the local node should perform outbound segment assembly and only deliver complete RUs. A value of 0x01 indicates that the application wishes the local node to deliver RU segments. See Segment Delivery.
Application pacing option	A value of 0x00 indicates that the application requires the local node to handle pacing. A value of 0x01 indicates that the application wishes to be involved with outbound pacing via Status-Resource messages. See Pacing and Chunking.
Application cancel option	A value of 0x00 indicates that the local node should automatically generate CANCEL. A value of 0x01 indicates that the application will generate CANCEL. See Inbound Chaining.
Application transaction numbers option	A value of 0x00 indicates that the application does not support transaction numbers. A value of 0x01 indicates that the application does support transaction numbers. See Recovery.
BIND check index	Gives the index of the BIND check table entry against which the BIND parameters should be verified. One of the following values should be used: 0x01 — 3270 printer session 0x02 — 3270 display session 0x10 — LUA (LU type 0) application

The **Open(PLU) Confirm** from the local node to the application indicates whether the BIND verification was successful, and if so, supplies the bind information control block (BICB). The BICB summarizes the session BIND parameters in a format suitable for high-level languages and effectively defines the characteristics of the PLU session. The application not negotiating the BIND should usually not require to examine the BIND on the Open(PLU) Request and should use the BICB on the Open(PLU) OK Confirm.

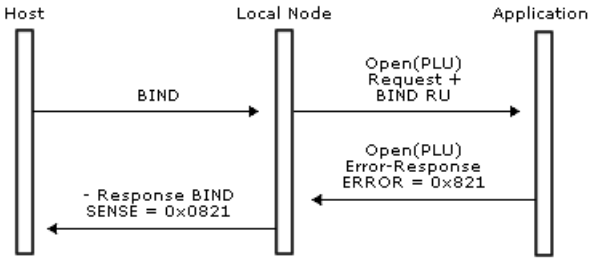
The following table summarizes the fields in the BICB and their correspondence to the parameters in the BIND RU. For more detailed information, see the IBM manual *Systems Network Architecture: Formats,* (GA27-3136). More detailed information is given in context in the topics dealing with PLU session characteristics such as pacing and brackets.

Bind Information Control Block Usage

Position on Open(PLU) OK Confirm	Position in Bind RU [byte,bit]	Description
dataru[0]	[2,]	FM profile
dataru[1]	[3,]	TS profile

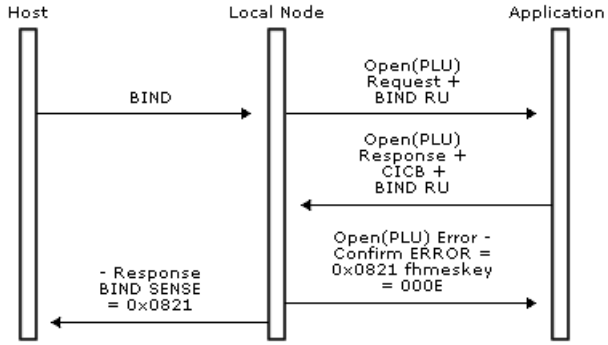
dataru[2]	[4, 0]	Primary chaining use
dataru[3]	[4, 1]	Primary request control mode
dataru[4]	[4,2-3]	Primary chain response protocol
dataru[5]	[4, 4]	Primary two-phase commit
dataru[6]	[4, 6]	Primary compression indicator
dataru[7]	[4, 7]	Primary send EB indicator
dataru[8]	[5, 0]	Secondary chaining use
dataru[9]	[5, 1]	Secondary request control mode
dataru[10]	[5,2-3]	Secondary chain response protocol
dataru[11]	[5, 4]	Secondary two-phase commit
dataru[12]	[5, 6]	Secondary compression indicator
dataru[13]	[5, 7]	Secondary send EB indicator
dataru[14]	[6, 1]	FM header usage
dataru[15]	[6, 2]	Bracket usage1
dataru[16]	[6, 2]	Bracket reset state2
dataru[17]	[6, 3]	Bracket termination rule
dataru[18]	[6, 4]	Alternate code set indicator
dataru[19]	[6, 5]	Sequence number availability
dataru[20]	[7,0-1]	Normal-flow send/receive mode
dataru[21]	[7, 7]	Half-duplex flip-flop reset
dataru[22]	[8,2-7]	Secondary pacing send window
dataru[23]	[9,2-7]	Secondary pacing receive window
dataru[24-25]*	[10,]	Secondary send maximum RU size
dataru[26-27]*	[11,]	Primary send maximum RU size
dataru[28]	[14,1-7]	LU-LU session type
dataru[29]	[27,]	PLU name size
dataru[30-37]	[28,]	PLU name (in EBCDIC)
dataru[38]	[15,0-3]	Session type 1: PS FMH type
dataru[39]	[15,4-7]	PS data stream profile
dataru[40]	[16, 0]	Number of outstanding destination s
dataru[41]	[16, 1]	Compacted data indicator
dataru[42]	[16, 2]	PDIR allowed indicator
dataru[43]	[15, 0]	Session type 2 or 3: query support
dataru[44]	[24,1-7]	Dynamic screen size
dataru[45]	[20,]	Basic row size
dataru[46]	[21,]	Basic column size
dataru[47]	[22,]	Alternate row size
dataru[48]	[23,]	Alternate column size
1 0x00 = Brackets not used 0x01 = Brackets used 2 0x01 = Bracket reset state is BETB (between-brackets) 0x02 = Bracket reset state is INB (in-bracket) *These values are of type INTEGER (all others are of type CHA R)		

The opening PLU sequence can fail if the application rejects the Open(PLU) Request (for example, if the BIND parameters are unacceptable on a nonnegotiable BIND) by sending Open(PLU) Error Response and appropriate sense codes. The local node sends to the host a negative response to the BIND request containing the supplied sense codes. The PLU connection is considered to be closed after an **Open(PLU) Error Response**, and the local node does not generate an **Open(PLU) Confirm**. The following illustration shows a failure to open the PLU connection (for a nonnegotiable BIND), due to the application rejecting the **Open(PLU) Request**.



- The opening PLU sequence can also fail if the BIND verification against the BIND check table entry specified by the application fails. In this case, the local node does the following: Sends to the host a negative response to the BIND request with appropriate sense codes.
- Sends to the application an Open(PLU) Error Confirm with the first word of the sense codes as the first error code and the index of the BIND parameter in error as the second error code.

The PLU connection is considered to be closed after the **Open(PLU) Error Confirm**. The following illustration shows failure to open the PLU connection due to BIND verification failure. Note that error code 2 gives the index in the RU of the BIND parameter in error.



Closing the PLU Connection

Either the application or the local node can terminate the PLU connection. The criteria for closing are:

- The local node closes the PLU connection if it receives an UNBIND request from the host PLU, which terminates the PLU session. If the UNBIND type is "BIND forthcoming" (0x02), the local node sets the BIND-forthcoming indicator in the Close(PLU) Request, so that the application can reserve any necessary resources.
- The local node closes the PLU connection if it receives a DACTLU or DACTPU request from the SSCP.
- The local node closes the PLU connection if it receives an outage notification from data link control.
- The local node closes the PLU connection if it detects a critical error in a message from the application, putting the application in a "critically failed" state. In this case the local node sends a TERM-SELF request to the host to elicit an UNBIND.
- The application should close the PLU connection for logical power-off conditions — for example, if its resources are temporarily unavailable, or when the user finishes using the session.

When the local node issues a Close(PLU) Request, the application can determine the reason from the Close control field. There may be an associated Status message on either the PLU connection (a Status-Acknowledge(Nack-2)) or the SSCP connection (a Status-Session message if the LU has been deactivated).

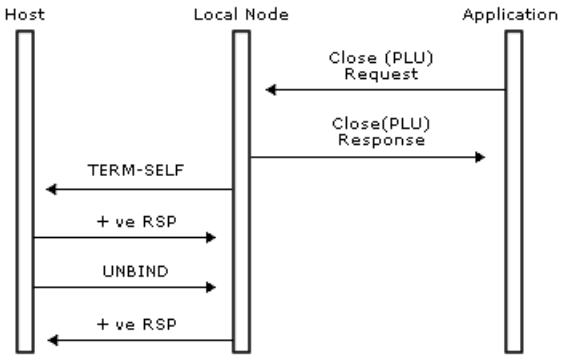
Whether the local node or the application closes the connection, the message is the same. The initiator of the Close sequence sends a Close(PLU) Request to its partner, which responds with a Close(PLU) Response. The Close(PLU) Request is unconditional — the Close(PLU) Response always reports that the connection was successfully closed.

The Close(PLU) Response is provided so that the initiator of the Close sequence can determine when outstanding data and status messages have been delivered. To avoid possible race conditions, the application should discard all messages it receives on the PLU connection after issuing a Close(PLU) Request, including any Close(PLU) Request messages from the local node, until it receives the Close(PLU) Response.

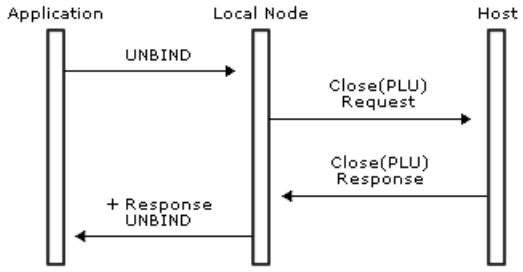
Note that, if the application sends a Close(SSCP) Request while the PLU session is active, the local node will close the PLU connection (as if Close(PLU) Request had been sent) as well as the SSCP connection.

The message sequence for an application-initiated Close is shown in the following figure. The local node sends a TERM-SELF request to the host to elicit an UNBIND.

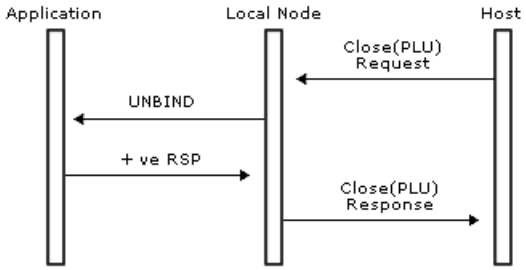
If the host generates an UNBIND automatically on receipt of a TERM-SELF, the application can view Close(PLU) as equivalent to the termination of the PLU-SLU session.



The message flow for a local node-initiated Close after receiving an UNBIND request from the host is shown in the following figure.



When an application is using the LUA variant of the FMI, issuing a Close(PLU) Request causes the node to immediately unbind the PLU session by sending an UNBIND request to the PLU. The **Close(PLU) Response** is returned to the application on receipt of the UNBIND response, as shown in the following figure.


Application
Local Node
Host
Close(PLU)
Request
UNBIND
+ ve RSP
Close(PLU)
Response

Using the PLU Session

When the PLU connection is open, the application has access to the PLU session and can communicate with the host PLU.

This section contains:

- PLU Session Characteristics
- PLU Session Status

PLU Session Characteristics

The local node provides support on the PLU session for FM profiles 2, 3, 4, and 7 and TS profiles 2, 3, 4, and 7. Support of these profiles means that the local node supports LU-LU session types 0, 1, 2, and 3. Using the PLU connection, the application can send and receive any FM data that is valid for the LU-LU types above.

The protocols appropriate to a particular session are determined by the parameters in the BIND request that establishes the session. The BIND parameters are reported to the application in the BICB on the Open(PLU) OK Confirm message. It is the application's responsibility to conform to the session protocols reported in the BICB.

Due to the wide range of BIND parameters allowable on a session and the options available to an application in the CICB on the Open(PLU) OK Response, this document does not attempt a complete description of the protocols for a particular session. The remaining topics in this section describe the general protocol characteristics of the PLU session, such as chaining, brackets, and so on.

Most of the protocol descriptions in the remainder of this section are accompanied by figures to illustrate the important features. The figures show:

- The relevant RH flags in SNA requests/responses.
- The sequence number of SNA requests/responses.
- Any sense data (shown as "SENSE=...") on SNA responses or Data messages.
- The ACKRQD field in Data and Status-Controlmessages.
- The relevant application flags (see Application Flags) in Data and Status-Control messages.
- The message key field in Data messages.
- Any error codes (shown as "ERROR=...") in Status-Acknowledge or Status-Control messages.

For simplicity, it is assumed that all messages are function management data flowing on the same PLU session that:

- Uses half-duplex flip-flop protocols.
- Uses brackets, with reset state of between-bracket.
- Does not use the PLU CICB segment delivery option (see Segment Delivery).

PLU Session Status

While the PLU connection is open, the local node reports any changes of state to the application via Status-Session messages. There is only one **Status-Session** status code that can occur on the PLU connection:

BETB	The PLU session has made the transition from the in-bracket state to the between-bracket state (see Brackets).

Status-Session Codes describes the **Status-Session** status codes.

Outbound Chaining

The local node checks that outbound chains of requests conform to the correct SNA usage, to the chaining usage for the session, and to the current state of the session. The local node will accept valid outbound chains of data from the host if one of the following is true:

- Data traffic is active on a full-duplex session.
- The session is in a state where it can receive data.
- The session is between brackets with neither half-session currently sending (see Brackets), or the session is in contention for a half-duplex contention session.
- The session is waiting for the host to initiate a recovery procedure (see Recovery); for example, the local node has sent a negative response to an outbound chain.

The local node sends a Data message to the application for each outbound request — but note the effects of the application specifying the segment delivery option in the connection information control block (see Segment Delivery). If the application does not specify segment delivery, then the BCI and ECI application flags in the message header reflect the chaining indicators in the RH of the request.

An outbound chain can terminate in several ways:

- The chain is received complete and without error; all the requests in the chain have been passed to the application as **Data** messages and have been acknowledged where applicable.
- The application detects an error in a Data message while receiving the chain. The application should send a Status-Acknowledge(Nack-1) with associated sense data to the local node, which sends a negative response plus the sense data to the host for the request corresponding to the Data message in error. The local node will not purge the remainder of the chain, so the application will see EC (end chain). Alternatively, the host can terminate the chain with a CANCEL, which is delivered to the application as a Status-Control(CANCEL) with ACKRQD set.
- The local node detects an error in a request and presents the application with a system detected error Data message to report the premature termination of the chain. This message carries the SDI (system detected error indicator) and ECI (end chain indicator) application flags, the sense codes for the error, and the ACKRQD indicator; it does not carry user data. When the application responds with Status-Acknowledge(Ack), the local node generates a negative response to the chain using the appropriate sense code. The application can use the reported sense codes to generate diagnostic information for the user (for example, a 3270 emulator would generate PROG check codes). The local node will purge the remainder of the chain, so the application may not see EC. Alternatively, the host can terminate the chain with a CANCEL, which is delivered to the application as a Status-Control(CANCEL) with ACKRQD set.
- The host can cancel the chain while sending, by sending the CANCEL request. The local node sends a Status-Control(CANCEL) message to the application, which the application must acknowledge.

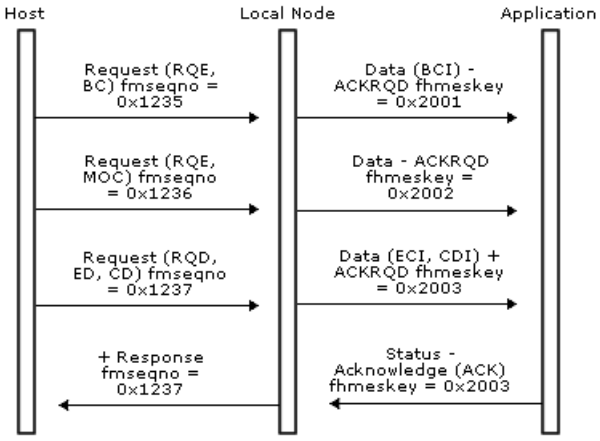
If an error occurs while receiving a chain, and the session uses half-duplex flip-flop protocols, then the application must assume an error-recovery-pending state (see Recovery).

For a session using half-duplex flip-flop protocols, if the application flags in the last Data message of the chain have the CDI (change direction) flag set, then:

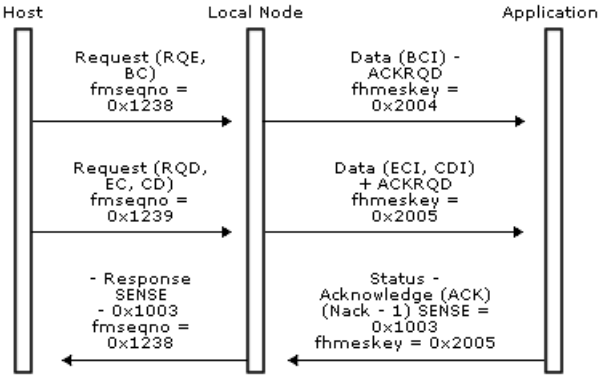
- If the chain was received without error, the application has direction.
- If the application rejected any message in the chain, the host retains direction.

The following four figures illustrate outbound chaining protocols between the local node and the application and how those protocols relate to the underlying SNA protocols.

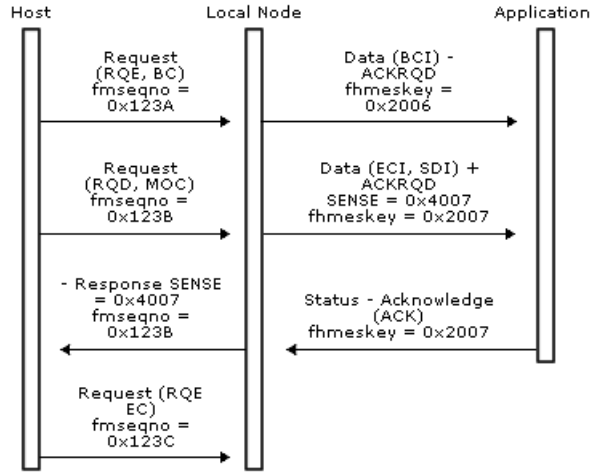
In the first illustration, a complete outbound chain is received without error and accepted by the application; note that after sending Status-Acknowledge(Ack) the application has direction.



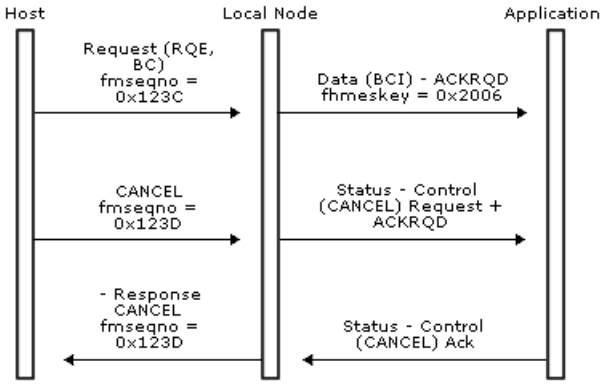
In the following illustration, a complete outbound chain is received without error, but is rejected by the application; note that even though the chain carried CD, the application does not have direction.



In the following illustration, the local node detects the invalid use of RQD without EC and converts the request to a **Data** message with the SDI application flag set, plus ACKRQD and appropriate sense codes. The application's **Status-Acknowledge(Ack)** drives the negative response to the host. This example assumes that the receive check 4007 has been specified in the CICB on the **Open (SSCP)**.



In the following illustration, the host cancels the outbound chain.


Host
Local Node
Application
Request (RQE, BC) fmseqno = 0x123C
Data (BCI) - ACKRQD fhmeskey = 0x2006
CANCEL fmseqno = 0x123D
Status - Control (CANCEL) Request + ACKRQD
- Response CANCEL fmseqno = 0x123D
Status - Control (CANCEL) Ack

Inbound Chaining

The division of application data into Data messages and the control of inbound chaining are the responsibility of the application.

The secondary maximum send RU size for the session is a parameter in the BIND from the host and is available in the BICB on the Open(PLU) OK Confirm message. The application should ensure that each inbound Data message corresponds to a single RU; that is, it does not contain more data than the maximum RU size given in the BICB.

The application should use the BCI and ECI application flags in the Data message headers to control chaining (see Application Flags). The chain is the unit of recovery, and if recoverable errors occur in the chain, then the application should assume responsibility for recovery (see Recovery).

An inbound chain can terminate in the following ways:

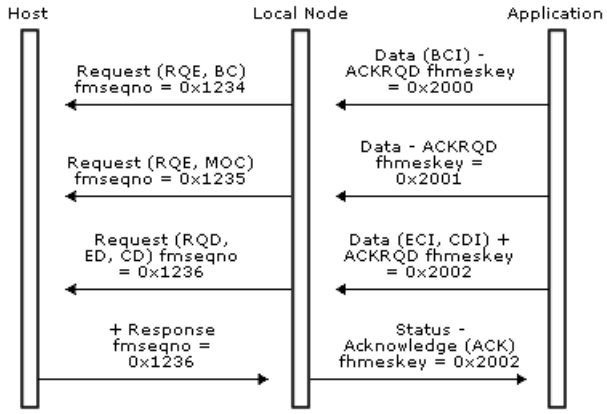
- The complete chain is sent without errors; that is, all the **Data** messages in the chain have been passed to the host. If the session allows the secondary to send definite-response chains, and the application sets the ACKRQD field in the last **Data** message of the chain, then the application receives a Status-Acknowledge(Ack) from the local node when the host supplies a response.
- The local node detects a critical error in the format of a Data message from the application or in the state of the session. The local node rejects the Data message with a Status-Acknowledge(Nack-2) containing an error code and closes the PLU connection. Note that the local node will generate an inbound CANCEL request before closing the PLU connection. The local node will send a TERM-SELF request to the host to elicit an UNBIND.
- The host sends a negative response to a request in the chain. The local node sends a Status-Acknowledge(Nack-1) message to the application with the sense codes and sequence number from the negative response. Where the host rejects a request that does not carry the ECI application flag, and the application did not specify the "application cancel," option in the PLU CICB, the local node also generates an inbound CANCEL request. When the application specifies "application cancel," then it must send EC or Status-Control(CANCEL) to terminate the chain. Any subsequent inbound chains are rejected with a noncritical Status-Acknowledge(Nack-2), sense code 0x2002 or 0x2004 (chaining or direction). When the application receives the Status-Acknowledge(Nack-1) message, it should stop sending data after this chain for half-duplex flip-flop sessions because the direction has passed to the host (see Direction).
- The application cancels the chain while sending, by sending a Status-Control(CANCEL) message to the local node. The local node sends a CANCEL request to the host and sends a Status-Control(CANCEL) Acknowledge to the application on receiving a positive response from the host. Responses from the host to requests sent before the CANCEL will generate appropriate Status-Acknowledge messages to the application if the original Data messages had the ACKRQD field set.
- The application closes the PLU connection while sending the chain. The local node sends a Close(PLU) Response to the application. Responses from the host to requests sent before the Close(PLU) message will not generate Status-Acknowledge messages to the application. Note that the local node will also generate an inbound CANCEL request and a TERM-SELF request to elicit an UNBIND.

If the local node detects a noncritical error in the format of a **Data** message from the application or the state of the session, it does not close the PLU connection. Instead, it rejects the **Data** message in error with a **Status-Acknowledge(Nack-2)** containing an appropriate error code. No data is sent to the host.

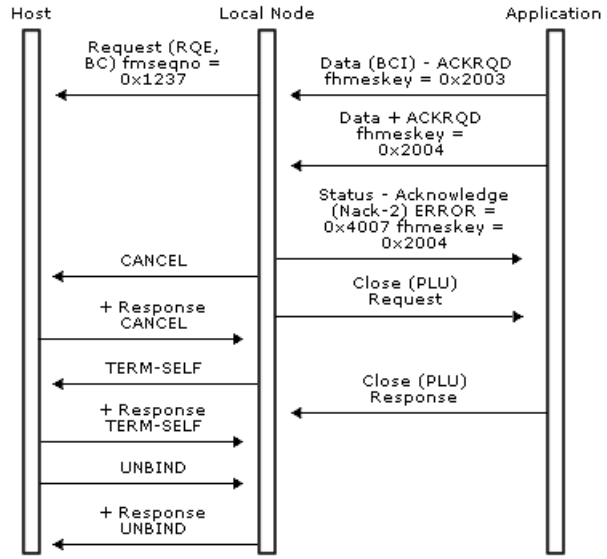
If an inbound chain terminates with an error, then if the session uses half-duplex protocols, the application must assume a receive state (see Recovery).

The following six figures illustrate inbound chaining protocols between the local node and the application, and how those protocols relate to the underlying SNA protocols.

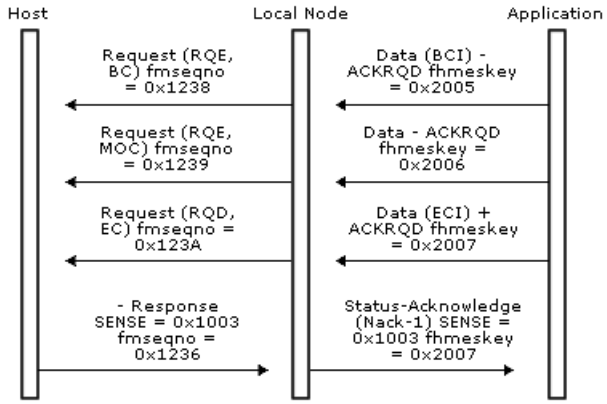
In the first illustration, a complete inbound chain is sent without error and accepted by the host; note that after receiving Status-Acknowledge(Ack) the application relinquishes direction to the host.



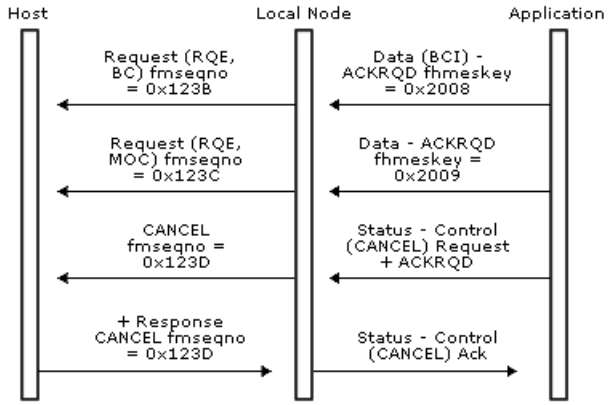
In the following illustration, the local node detects a critical error in the format of the second **Data** message in the chain (ACKRQD without the ECI application flag), sends a **Status-Acknowledge(Nack-2)** to the application with the appropriate error code, and closes the PLU connection. Note that the local node only generates the CANCEL where the session's FM profile supports CANCEL.



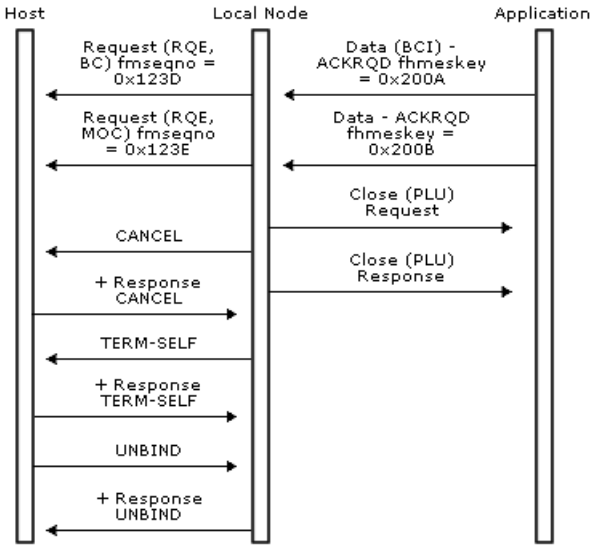
In the following illustration, a complete inbound chain is sent without error, but is rejected by the host; after the negative response, the application must enter receive state, pending error-recovery (see Recovery).



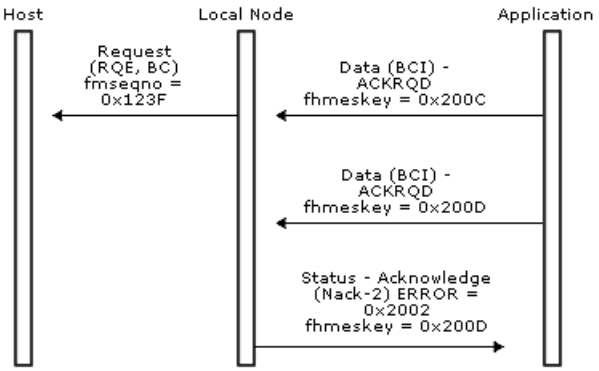
In the following illustration, the application cancels the chain by sending **Status-Control(CANCEL)**; note that the application still has direction and can start a new chain.



In the following illustration, the application closes the PLU session while sending the chain; the local node only generates the CANCEL where the session's FM profile supports CANCEL.



In the following illustration, the local node detects a noncritical error in the format of the second **Data** message in the chain and sends a **Status-Acknowledge(Nack-2)** to the application with the appropriate error code.



Segment Delivery

Where the maximum RU size for a session (supplied in the BIND parameters) allows RUs that are larger than the maximum size of a data link control transmission unit (for example, an SDLC frame), then the local node's path control is responsible for assembling outbound segments into RUs and segmenting inbound RUs where required.

However, certain IBM products (for example, SNA models of the 3270 controllers) do not perform outbound segment assembly, to improve perceived response times at display terminals by displaying each segment as soon as it is received. This feature is referred to as "window shading."

The local node allows an application to specify a segment delivery option in the CICB on the Open(PLU) OK Response. If an application specifies this option, the local node's path control does not assemble outbound segments into complete RUs, and the local node delivers the segments to the application in Data messages. This allows an application emulating a 3270 device to reproduce the perceived response characteristics of the IBM device. In cases where throughput is high, such as 3270 file transfer, segment delivery can give improved performance compared to RU delivery.

Note that there is no comparable feature for inbound data — the application must present Data messages containing complete RUs to the local node. Also, there is no support for segment delivery on the SSCP session and connection (where the maximum RU size is limited to 256 bytes).

The local node supports the segment delivery option in such a way that the constraints placed on an application receiving data in either form are identical. If complete RUs are required, then the local node rebuilds the RUs from segments in path control. If segments are required, the local node handles all segmentation indicators and modifies processing within its SNA layers to cater for segmented RUs.

All Data messages delivered to the application contain application flags, whereas only the first segment in an RU contains an RH. The local node delays the EC and CD indicators if they occur in the RH of the RU's first segment, and sets the corresponding ECI and CDI application flags in the Data message corresponding to the last segment of the RU. Therefore the Data messages corresponding to RU segments have application flags set as if they corresponded to whole RUs. This considerably simplifies the handling of chaining, bracket, and half-duplex protocols for an application using the segment delivery option.

Note that EB is not delayed until EBIU, since the application should use the Status-Session between-brackets message to determine when to enter the between-brackets state.

Brackets

This section primarily describes the bracket protocols between the local node and an application for a session supporting half-duplex flip-flop with brackets.

The local node enforces no bracket protocols for full-duplex sessions; therefore, messages with BB are not presented as Status-Control(BID) messages, and there are no Status-Session(BETB) messages.

The management of this protocol for a generalized application is complex, and there is a significant amount of code in the local node to simplify the application's perception of the protocol. An application is only aware of two states:

- In-bracket
- Between-bracket

The local node, in addition to the states of in-bracket and between-bracket, maintains transient states with a large state transition matrix, or finite-state machine, governing the half-session's state at a particular time.

This section contains:

Bracket Initiation

While a session is in the between-bracket state, contention exists. Either the application or the host PLU can attempt to initiate a bracket, as follows:

- The application initiates a bracket by sending a Data message with the BBI application flag and ACKRQD set while in the between-bracket state. The local node sends a request corresponding to the **Data** message to the host PLU. The application has successfully initiated a bracket and is in the in-bracket state. Flip-flop protocols are now in force until the bracket is terminated.
- The application initiates a bracket by sending a Status-Control(LUSTAT) with the BBI application flag set while in the between-bracket state. The local node sends an LUSTAT request to the host PLU. The application has successfully initiated a bracket and is in the in-bracket state. Flip-flop protocols are now in force until the bracket is terminated.
- The host PLU sends a BID request while in the between-bracket state. The local node sends a Status-Control(BID) with ACKRQD to the application (see Status-Control Message). The application replies with a Status-Control(BID) Acknowledge, to indicate that it is willing to accept a bracket. The local node sends a positive response to the BID request. The host PLU has successfully initiated a bracket, and the application's state is in-bracket, with flip-flop protocols applying until the bracket is terminated.
- The host PLU sends data in an RU carrying the BB indicator in the RH while in the between-bracket state. The local node presents this method of initiating a bracket in the same way as if the host PLU had initiated the bracket with BID. The local node sends a Status-Control(BID) with ACKRQD to the application. The application replies with a Status-Control(BID) Acknowledge to indicate that it is willing to accept the bracket. The local node sends the Data message corresponding to the RU to the application and sends a positive response to the data RU. The host PLU has successfully initiated a bracket, and the application's state is in-bracket, with flip-flop protocols applying until the bracket is terminated.
- The host PLU sends an LUSTAT request carrying the BB indicator in the RH. The local node presents this method of initiating a bracket in the same way as if the host PLU had initiated the bracket with BID. The local node sends a Status-Control(BID) with ACKRQD to the application. The application replies with a Status-Control(BID) Acknowledge to indicate that it is willing to accept the bracket. The local node sends a Status-Control(LUSTAT) to the application, which requires an acknowledgment. The host PLU has successfully initiated a bracket, and the application's state is in-bracket, with flip-flop protocols applying until the bracket is terminated.
- The host attempts to initiate a bracket using a BID request or an RU carrying BB, which the local node sends to the application as a Status-Control(BID), but the application cannot accept the bracket. The application should send a negative Status-Control(BID) response with an appropriate sense code. The local node sends a negative response to the BID carrying the sense code supplied by the application. The application's state is still between-bracket. The application should use one of the following sense codes:

  0x081B if it has already committed resources for an inbound transfer; for example, a terminal operator has begun typing.

  0x0814 if it currently cannot begin a bracket but will notify the host when resources become available; for example, a 3270 printer is being used for local copy in between-bracket printer sharing mode. At a later stage when the resources become available, the application should temporarily reserve the resources and send a Status-Control(RTR) to the local node. If the host rejects the RTR, the local node returns a Status-Control(RTR) Negative-Acknowledge-1 response, and the application can release the resources. Otherwise, the host attempts to initiate a bracket that the application must now accept.

- Where the application has successfully initiated a bracket, a bracket race may occur due to the host PLU attempting to initiate a bracket. The application gets a **Status-Control(BID) Request**, which it should reject with 0x080B or 0x0813. The application retains direction after race negative responses (see Recovery). The application's bracket state remains as in-bracket.

The application needs to be aware of one further complication in bracket initiation. All the above cases relate to sessions whose bracket reset state is between-bracket; that is, a state of contention exists, and either half-session can attempt to begin a bracket.

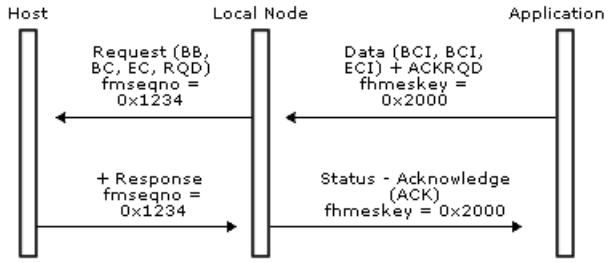
However, the BIND parameters for the session can specify a bracket reset state of in-bracket. Where the bracket reset state is in-bracket, one half-session is considered to have already successfully initiated a bracket. Flip-flop protocols will then apply until a Status-Session(BETB) is received, at which time the session reverts to a contention state and bracket initiation proceeds as described above.

The application must set its bracket state when the PLU connection is opened (that is, on receipt of the Open(PLU) OK Confirm message) and reset it each time the session is reset (that is, after receipt of a Status-Control(CLEAR) Request). The appropriate bracket reset state for the session is supplied to the application in the BICB on the Open(PLU) OK Confirm message.

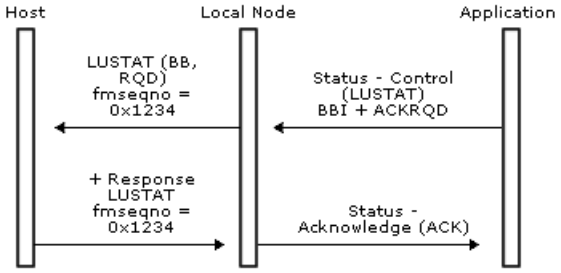
The following six figures illustrate bracket initiation protocols between the local node and the application and how those protocols

relate to the underlying SNA protocols.

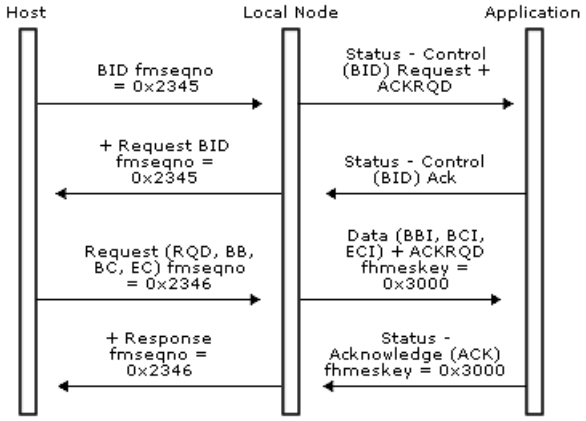
In the first illustration, the application initiates a bracket by sending an inbound chain with the BBI application flag set when its state is between-bracket. The application's state is in-bracket until it receives a Status-Session(BETB). (If the application can send RQE chains, a bracket can be opened by sending an RQE chain.)



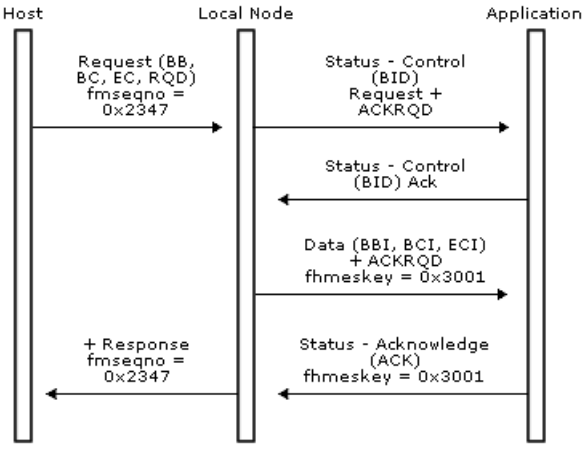
In the following illustration, the application initiates a bracket by sending a **Status-Control(LUSTAT)** with the BBI (begin bracket indicator) application flag set when its state is between-bracket. The application's state is in-bracket until it receives a **Status-Session(BETB)**. The LUSTAT can be sent NOACKRQD (that is, RQE) if required.



In the following illustration, the host initiates a bracket by sending BID, which the application accepts. The application's state is in-bracket and the host can send.

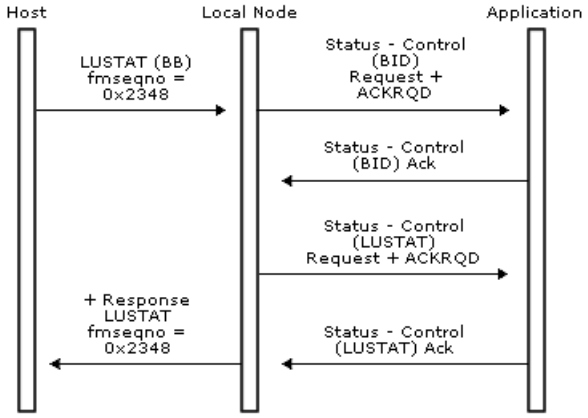


In the following illustration, the host PLU initiates a bracket by sending a request with BB (begin bracket), which the application accepts. The application's state is in-bracket, and the host can send.

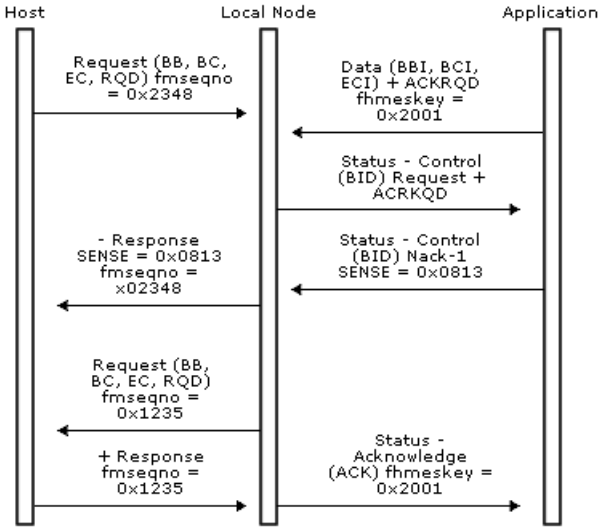


In the following illustration, the host initiates a bracket by sending an LUSTAT with BB, which the application accepts. The

application's state is in-bracket, and the host can send.



In the following illustration, the host and application both attempt to initiate a bracket in between-bracket state; the application rejects the host bids with sense code 0x0813, and the local node delivers the application's data. After sending the data, the application's state is in-bracket, and the application can send.



Bracket Termination

The local node supports bracket termination rule one (conditional) and bracket termination rule two (unconditional), as specified in the BIND request. Some sessions only allow bracket termination by one session partner; this is a BIND option (supplied in the BICB on Open(PLU) OK Confirm), and it is the application's responsibility to determine if (and when) it should request bracket termination.

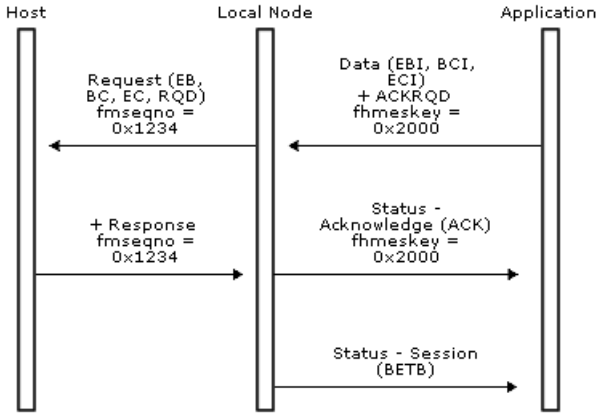
If an application is allowed by its BIND to terminate brackets, it does so by setting the EBI application flag in an inbound Data or Status-Control(LUSTAT/CHASE/QC/CANCEL) message. The bracket is only terminated when the application receives a Status-Session (BETB) from the local node.

If the host terminates a bracket successfully, the local node sends a Status-Session(BETB) to the application. Note that the EBI application flag on outbound messages does not indicate bracket termination, but indicates that the corresponding RU carried EB. The bracket is only terminated when the application receives Status-Session(BETB).

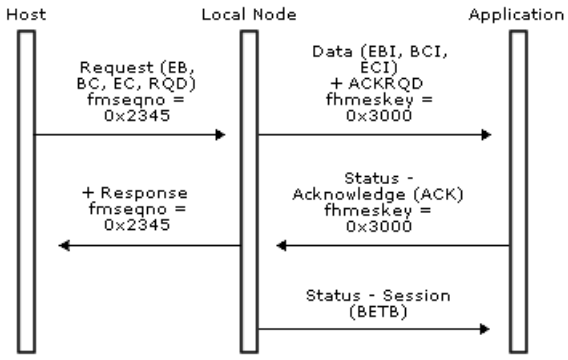
Note that if the application queues data, then it should also queue Status-Session(BETB) messages; they must not be processed as expedited.

The following two figures illustrate bracket termination protocols between the local node and the application and how those protocols relate to the underlying SNA protocols.

In the following illustration, the application successfully terminates a bracket by sending an EBI data chain when the application's state is in-bracket, which the host accepts. The local node sends a Status-Session(BETB) to indicate that the application's state is now between-bracket.



In the following illustration, the host successfully terminates a bracket by sending an EBI data chain when the application's state is in-bracket, which the application accepts. The local node sends a **Status-Session(BETB)** to indicate that the application's state is now between-bracket.



Direction

When an FMI application is communicating on its PLU connection with a normal flow request mode other than full-duplex (that is, half-duplex flip-flop or half-duplex contention), it must obey the SNA direction protocol. These two modes are treated separately.

This section contains:

Half-Duplex Flip-Flop Direction

The BIND used to establish the session carries information about the initial state of the bracket and direction machines. This can be specified in the BIND if either of the following conditions are satisfied:

- Brackets are not used.
- Brackets reset state is in-bracket.

If neither of the conditions hold, then the initial direction state is contention.

When the direction is specified in the BIND, the application should assume the direction state specified in the half-duplex reset state as soon as data can flow. This field can be obtained indirectly by using a BIND check index that only accepts a particular direction, or directly by reading the HDXRSET field in the BICB on the Open(PLU) OK Confirm message (see Opening the PLU Connection), or by reading the BIND on the Open(PLU) Request.

When in contention state, either the PLU or the application can initiate a bracket (see Brackets); the successful initiator of the bracket obtains direction (unless direction is relinquished when opening the bracket by sending BB, BC, EC, or CD). Since the secondary is assumed to be the contention winner, the application can assume send state from contention sending BB and rejecting any subsequent Status-Control(BID) Request from the local node before receiving Status-Session(BETB). When the application accepts a Status-Control(BID) Request in contention state, then it must assume receive state.

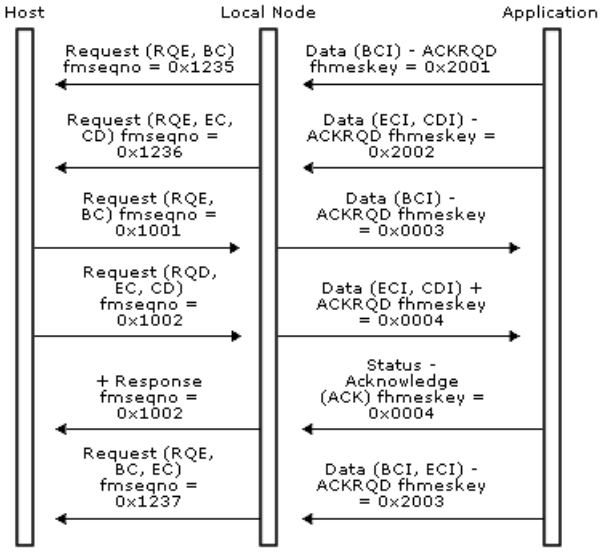
Half-duplex flip-flop direction can change through the following actions:

- Sending or receiving data with the change direction (CD) indicator in the RH (and the corresponding CDI (change direction indicator) flag on the DATAFMI and Status-Control messages). Note that CD is only used at the end of a chain (and for applications receiving segments will be delivered with ECI, EBIUI). Also note that CD is valid on the following normal flow **Status-Control** requests: LUSTAT, CANCEL, CHASE and QC.
- Receiving a negative response when the application should assume receive state (error recovery pending state — see Recovery).
- If the application rejects data from the host carrying CDI, then it must remain in receive state.

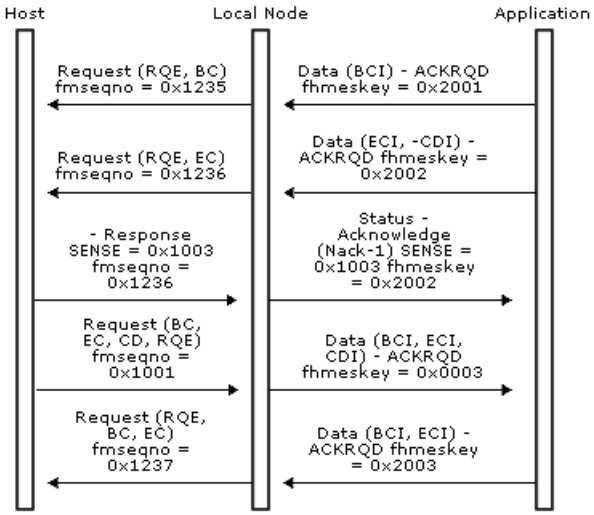
Providing the FM profile is correct (3, 4, or 7), the application can request direction from the host using a **Status Control(SIGNAL) Request** with CODE1 set to 0x0001; CODE2 is set to a user-defined value.

The following three figures illustrate the direction protocol for applications using the half-duplex flip-flop mode.

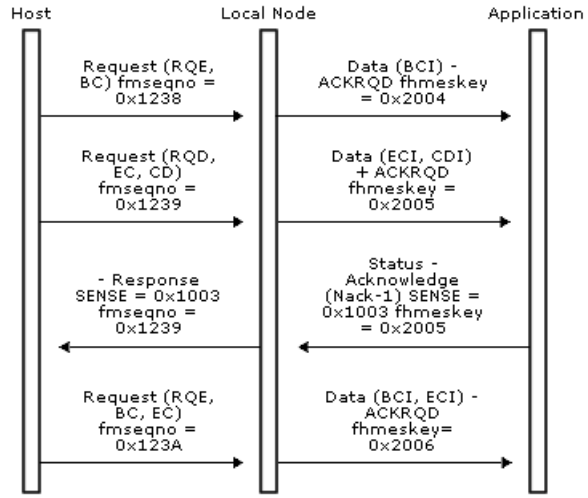
In the first illustration, the application issues and receives the CD without error.



In the following illustration, the host sends a negative response to inbound data; the application assumes receive state, and then the host sends CD to give the application direction.



In the following illustration, a complete outbound chain is received without error, but is rejected by the application; note that even though the chain carried CD, the application does not have direction.



Half-Duplex Contention

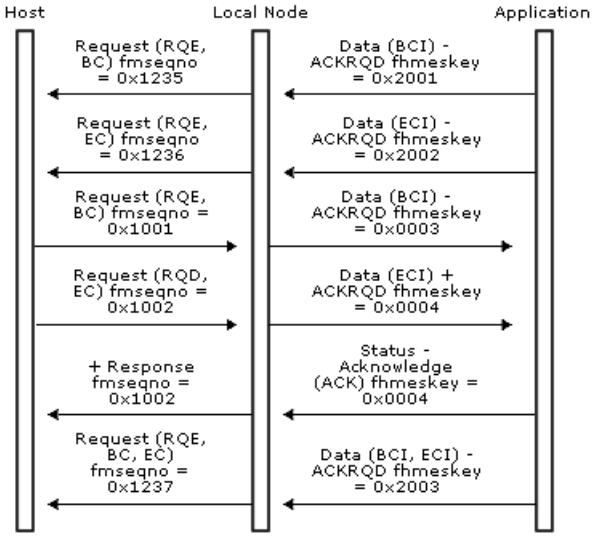
For half-duplex contention, the initial direction state is contention. Half-duplex protocol operates during a chain (only one partner can send), but the direction state normally returns to contention at the end of each chain. The CD indicator in the RH is thus not required; however, if the CD indicator is used, direction is reserved for the receiving half-session. Therefore, if the application receives CD, it should assume send state and not expect to receive data. Conversely, if the application sends CD, then it cannot send again until it has received a chain from the host.

In the event of an error being discovered by either half-session, the application must assume receive state, since the host is responsible for recovery.

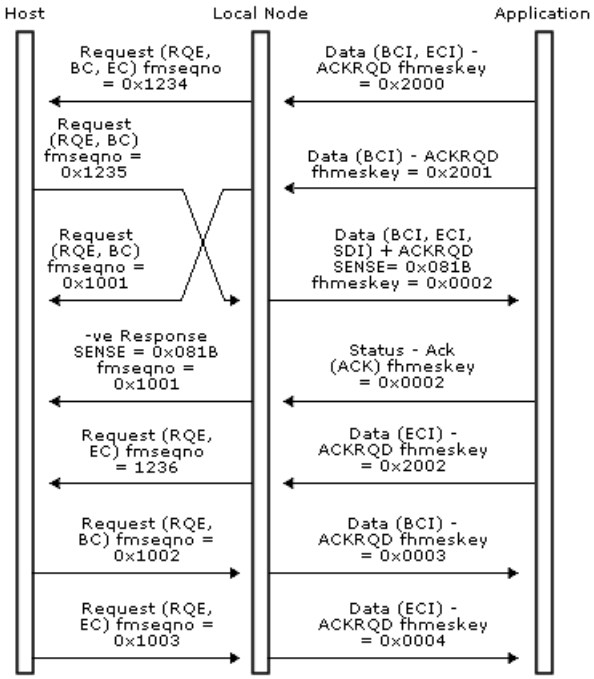
If both half-sessions attempt to start a chain when the direction state is contention, the race is resolved in favor of the secondary application using a sense code of 0x081B. However, the possible window between the local node and the application means that the local node cannot determine when outbound RQE data is received by the application. Therefore, if the local node receives data from the application while it determines that the half-duplex contention state is receive, it will reject it with a noncritical NACK-2 (0x2004 direction).

The following two figures illustrate the direction protocol for applications using half-duplex contention mode. The three figures in the previous topic would also be valid although CD does not need to be specified.

In the following illustration, the application issues and receives data using half-duplex contention protocol without error.



In the following illustration, the half-duplex contention race is resolved in favor of the application.



Pacing and Chunking

The local node supports session pacing inbound and outbound, according to the pacing values in the BIND parameters for the session. The application can be involved in outbound pacing through the use of the Status-Resource message; inbound pacing is handled transparently by the local node and need not concern the application.

This section contains:

Outbound Pacing

If the application has enough resources to handle outbound data as fast as the network can provide it (for example, a screen), or if a higher level protocol (for example, immediate request mode) constrains the data flow, then the application need not be involved in pacing, and it is possible for the local node to handle outbound pacing transparently.

However, certain types of applications may require involvement in outbound pacing. If the application has limited resources (for example, a printer), then the application should specify the application pacing option in the CICB on the Open(PLU) OK Response (see Opening the PLU Connection) and provide the local node with information on the state of these resources initially on the Open(PLU) OK Response and periodically using Status-Resource messages.

To assist the application in calculating the initial credit field in the Open(PLU) OK Response, the local node delivers the pacing window sizes and the primary and secondary maximum RU sizes on the Open(PLU) Request. The initial credit must be at least as large as the primary to secondary pacing window size; otherwise the BIND will be rejected and the application will be sent an Open(PLU) Error Confirm message. The local node fills in a suggested initial credit value of one more than the pacing window (to try to avoid stop-start situations).

Note that the local node will also reject the BIND if the application specifies that it wishes to be involved in pacing (of whatever initial credit), but the BIND specifies that there is no outbound pacing.

Only FMD (function management data) requests are part of the credit scheme, so the application must maintain space within its buffer for one Status-Control request per RU in addition to the number of RUs specified by the initial credit count (a Status-Control message takes up 36 bytes).

Each unit of credit that the application delivers to the local node allows the local node to give the application a single RU (or a single chunk if chunking is being used). Note that if the application is receiving segments, then this may correspond to several DATAFMI messages. The application can count RUs for the purpose of outbound flow control by using the BBIU (begin basic information unit) and EBIU (end basic information unit) flags.

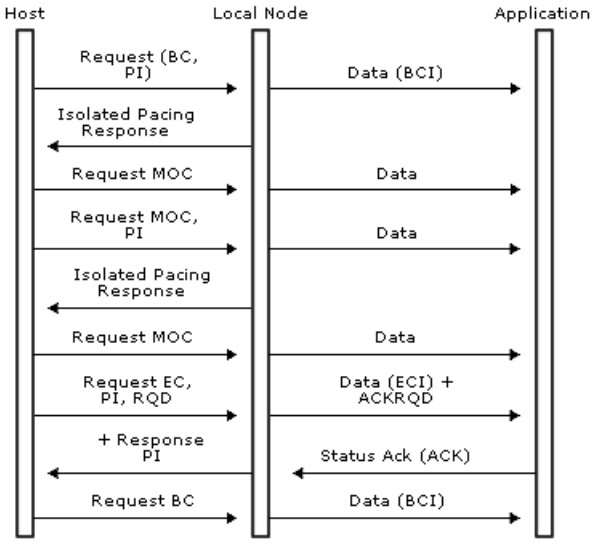
The application should maintain a credit-used count, which it should report to the local node on Status-Resource messages. The application needs to take the following actions:

- On processing (not receiving) DATAFMI messages with EBIU set (corresponding to FMD requests), increment the credit-used count by one.
- On processing Status-Control messages and all other messages from the local node, do not increment the credit-used count.
- Periodically report the current credit-used count on a Status-Resource message.
- Report the credit-used count when its buffer becomes empty (whatever the last message processed was), unless the credit-used count is zero.
- When the credit-used count is reported to the local node, reset it to zero.

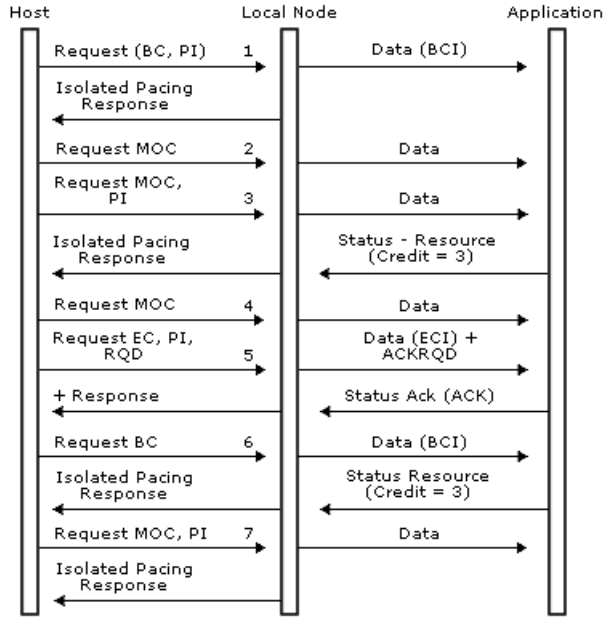
The frequency at which the application provides Status-Resource messages is not architected. However, the local node will only send the application as many Data messages as it has received credit for, and so when the application's credit-used count reaches the initial credit value, the local node will not send any more data. The application should attempt to send a **Status-Resource** message before this happens, because if the local node cannot send a **Data** message to the application and the host is still sending requests, the local node may not be able to send a pacing response to the host when required, with a consequent degradation of performance.

If the pacing window is small, such as one or two, the application should send a Status-Resource after processing each DATAFMI message to allow the local node to send the suitable pacing response.

The following figure shows the local node handling outbound pacing when the application is not involved (APPLPAC = 0x00); the pacing window is assumed to be two.



The following figure shows the local node and the application handling outbound pacing with the outbound pacing window assumed to be two and the initial credit from the local node to the application assumed to be four. Note that the local node can send an isolated pacing response (IPR) to the host to get another window full of data as soon as the application has sufficient credit for the rest of the present window and the next window.



Chunking

> Previous versions of this document indicated this as a future feature. The support is enabled in Comm Server 1.2 (and later) under OS/2, and is supported in Microsoft Host Integration Server and Microsoft SNA Server for Microsoft Windows 2000, Windows NT, Windows 98, and Windows 95. Applications should therefore test the product version returned on a call to sepdgetinfo for version 1.2 or later before using the chunking system.

In some cases, the RU size used by the local node may be too large for the length of the path between the local node and an FMI application; for example, when using a 16MB token-ring link, which can support 16KB frames. The local node allows an FMI application to specify that data transfer should be in smaller units, called chunks.

For example, consider an implementation using OS/2 named pipes, with a maximum pipe size of 4K. If the maximum RU size is 8K, the application can specify that data is to be transferred in smaller chunks of 1K. Together with a credit limit, this ensures that the pipe will never be completely full. (It is important to ensure that the pipe is never filled, because if the local node attempts to write to a full pipe, it will be blocked, causing severe performance problems.)

Chunking can be thought of as similar to segmentation (see Segment Delivery for more information); the distinction is that segmentation is determined by the communications link between the local node and the remote system, whereas chunking is determined by the communications link between the application and the local node.

The application indicates on the Open(SSCP) Request whether it supports chunking, and, if so, the chunk size in bytes that it wishes to use. The local node then uses the RU size, the chunk size, and the segment size (if applicable) to determine whether chunking is necessary. It then specifies the chunk sizes used for inbound and outbound flow (which need not be the same) on the Open(PLU) Request; these values are specified in units of elements (see Messages for more information). A value of zero for either of these sizes indicates that chunking is not necessary because the chunk size is not the limiting factor. Note that in chunking data, an RU will not be split in the middle of an element; this avoids data copying.

For example, assume that the local node is using an RU size of 8K and segments of 2K, and the application's Open(SSCP) Request specifies segment delivery and a chunk size of 4K. Chunking will be used on inbound data flow (because the chunk size is smaller than the RU size), but is not necessary on outbound data flow (because data will be delivered in segments that are smaller than the chunk size).

If chunking is being used in either direction, then all credit values specify the number of chunks that can be sent in that direction, not the number of RUs. Note that the segment delivery option is included on the Open(SSCP) Request to allow the local node to calculate the initial chunk credit values on the corresponding PLU connection. The application must also set this option on the Open(PLU) Response; if the Open(SSCP) Request and the Open(PLU) Response have different settings of this option, the setting from the Open(PLU) Response will be used. This can mean that the initial credit value used is not appropriate.

If session-level pacing is being used, the local node links this to the chunking credit. In particular, if the application withholds credit, the local node will delay sending a pacing response to the host, thereby applying back pressure to the host. This linkage is handled by the local node and need not concern the application.

Application flags (see Application Flags) on chunks of RUs are handled in the same way as those on segments (see Segment Delivery for more information). In particular:

- FMHI, BCI, COMMIT, BBI, EBI, CODE, ENCRYP, ENPAD, QRI, and CEI are only set on the first chunk of an RU.
- ECI and CDI are only set on the last chunk of an RU.
- BBIUI is always set on the first chunk of an RU.
- EBIUI is always set on the last chunk of an RU.

Note that EBI is set on the first chunk of the last RU in a bracket and not on the last chunk as might be expected; this is the same behavior as for segment delivery. The application should use the **Status-Session(BETB)** message, not the EBI flag, to determine when a bracket has ended.

Chunks are identified using the segmentation flags BBIUI and EBIUI, and therefore the application cannot distinguish between chunks and segments if both segmentation and chunking are being used outbound. However, there is generally no need for the distinction; the application can perform window shading (see Segment Delivery for more information) by displaying each unit of data as it is received, whether the unit of data is a segment or a chunk.

Confirmation and Rejection of Data

The following topics describe conditions under which inbound and outbound data is confirmed or rejected.

This section contains:

Confirmation and Rejection of Inbound Data

For every SNA chain of data sent or received for which responses are outstanding (RQE or RQD), the local node maintains a correlation table entry (CT). If the CTs become depleted, then the local node will terminate the session using the most CTs. A Status-Error message (code 0x46) and a Close(PLU) Request are sent to the application, and a TERM-SELF message is sent to the host. CT shortages (inbound) can be avoided by sending CD (for half-duplex) data, or data ACKRQD, or any **Status-Control(CHASE)**, or **Status-Control(LUSTAT)** with ACKRQD. Outbound shortages can be avoided by sending courtesy acknowledge messages as described in Opening the PLU Connection.

The local node sends chains of data to the host with their chain response mode specified as follows:

1. Definite

   If the application sends a **Data** message to the local node with the ACKRQD field set, and the BIND parameters specified that the secondary uses definite or definite/exception response mode.

2. Exception

   If the application sends a **Data** message to the local node without the ACKRQD field set, and the BIND parameters specified that the secondary uses exception or definite/exception response mode.

3. No-Response

   If the application sends a **Data** message to the local node without the ACKRQD field set, and the BIND parameters specified that the secondary uses no-response mode.

If the setting of ACKRQD on a Data message from the application does not reflect the chain response mode specified in the BIND parameters, the local node returns a Status-Acknowledge(Nack-2) indicating a noncritical error code; for example, if the application specifies ACKRQD but the BIND parameters do not permit the local node to send definite response chains.

In case 1, the application receives an acknowledgment to all FMD chains it sends to the local node:

- Positive responses from the host are returned to the application as Status-Acknowledge(Ack) messages.
- Negative responses from the host are returned as Status-Acknowledge(Nack-1) messages carrying the SNA sense codes.
- Errors detected by the local node when attempting to send the message are returned as Status-Acknowledge(Nack-2) messages carrying the equivalent error code.

In case 2, the application only receives an acknowledgment of an FMD chain it sends to the local node for:

- Negative responses from the host, which are returned as **Status-Acknowledge(Nack-1)** messages carrying the SNA sense codes.
- Errors detected by the local node when attempting to send the message, which are returned as Status-Acknowledge(Nack-2) messages carrying the equivalent error code.

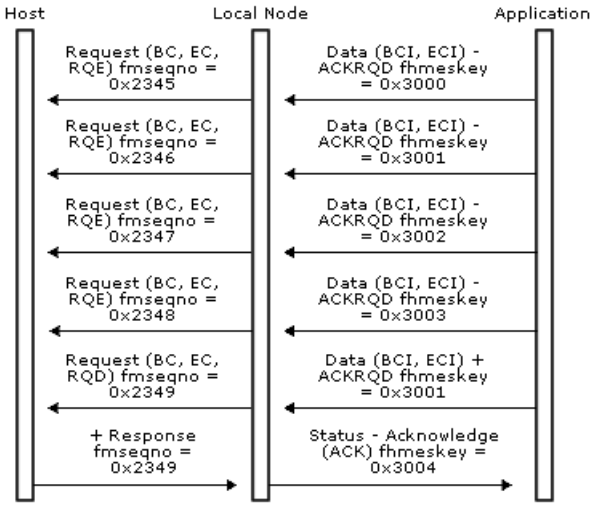
In case 3, the application only receives an acknowledgment of an FMD chain it sends to the local node when the node detects an error in the message and sends the application a **Status-Acknowledge(Nack-2)**. The only dissent that the host can make is to send a subsequent LUSTAT 0x400A (no response not supported) with the sequence number of the request in the sense qualifier field; this is presented to the application as a **Status-Control(LUSTAT)** as usual.

Whenever an application receives a Status-Acknowledge(Ack) or Status-Acknowledge(Nack-1), it implicitly confirms receipt by the partner half-session in the host of all previously sent chains.

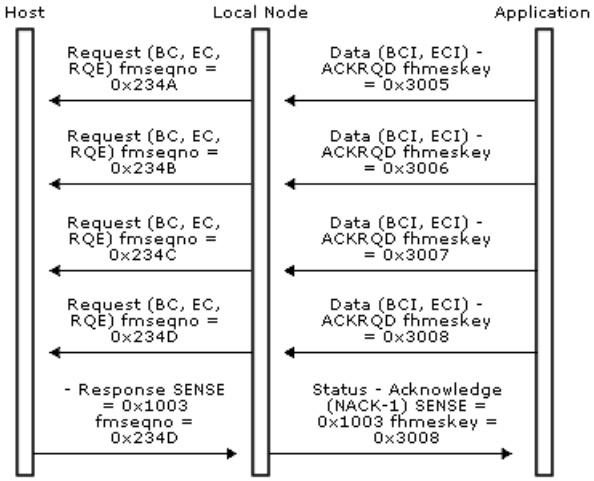
In case 2, the application does not usually receive such responses from the host to chains it has sent, and in case 3, the application never receives such responses. Therefore, to get the host to confirm receipt of all previously sent chains, the application should issue a Status-Control(CHASE) Request with ACKRQD set. This causes the local node to generate an SNA CHASE request to the host. The receipt of the response to this CHASE confirms that the host has received this CHASE request and all previous chains sent by the application. The local node issues a Status-Control(CHASE) Acknowledge to notify the application that this is so.

The following three figures illustrate the inbound data confirmation and rejection protocols between the local node and the application, and how those protocols relate to the underlying SNA protocols.

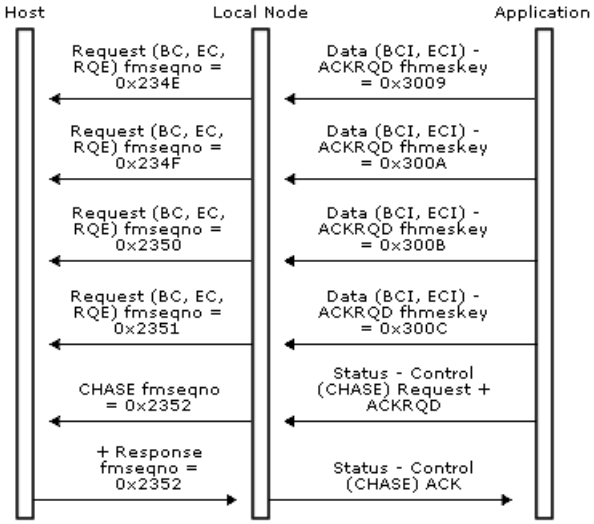
In the first illustration, an application sets the ACKRQD field in an inbound data chain to get the host to confirm receipt of the chain and all previously sent chains.



In the following illustration, the **Status-Acknowledge(Nack-1)** rejects the last chain, but confirms receipt by the host of all previously sent data chains.



In the following illustration, the application uses a **Status-Control(CHASE)** to get the host to confirm receipt of the corresponding CHASE request and all previously sent chains.



Confirmation and Rejection of Outbound Data

The local node sends chains of data from the host to the application with their ACKRQD field set as follows:

1. ACKRQD set

   If the corresponding SNA request was received specifying definite response, and the BIND parameters specify that the primary uses definite or definite/exception chain response mode.

2. ACKRQD not set, response mode

   If the corresponding SNA request was received specifying exception response, and the BIND parameters specify that the primary uses exception or definite/exception chain response mode.

3. ACKRQD not set, no-response mode

   If the corresponding SNA request was received specifying no response, and the BIND parameters specify that the primary uses no-response chain response mode.

In case 1, the application should always send an acknowledgment as follows:

- If the application accepts the data, it should return a Status-Acknowledge(Ack) message.
- If the application wishes to reject the data, it should return a Status-Acknowledge(Nack-1) message carrying the appropriate SNA sense codes.

In case 2, the application should only send an acknowledgment in the following cases:

- If the application wishes to reject the data, it should return a **Status-Acknowledge(Nack-1)** message carrying the appropriate SNA sense codes.
- The application can send a courtesy acknowledgement (see Outbound Data) to an RQE message to clear up correlation data within the local node.

In case 3, the application should not send acknowledgments; however, the sending of a **Status-Acknowledge(Ack)** or **Status-Acknowledge(Nack-1)** by the application has no effect — it is discarded.

Whenever an application sends a Status-Acknowledge(Ack) or Status-Acknowledge(Nack-1) to a received Data message, it implicitly confirms receipt of this and all previously-received Data messages.

In case 2, the host can issue a CHASE request; the local node sends a Status-Control(CHASE) Request with ACKRQD set to the application. When the application is in a position to confirm receipt of all outstanding data, it should issue a Status-Control(CHASE) Acknowledge message, which the local node converts into a positive response to CHASE for the host.

In cases 1 and 2, if the local node detects an error in a received request, it converts the request into a special Data message, which it passes to the application. Regardless of the chain response mode specified for the secondary in the BIND parameters, this Data message has the following characteristics:

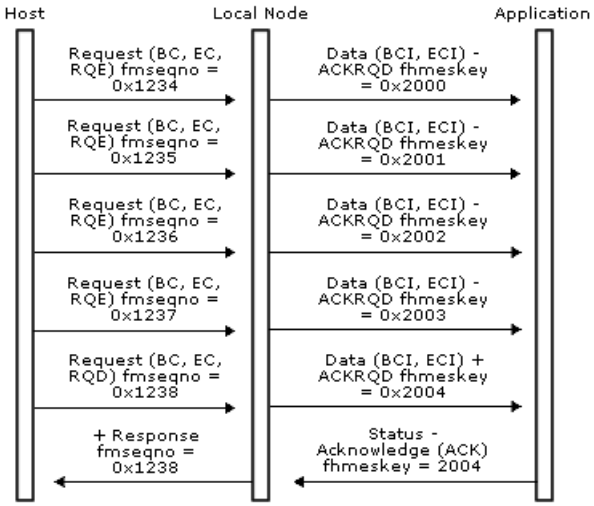
- ACKRQD is set; that is, the application must confirm receipt using a Status-Acknowledge(Ack) message.
- The SDI application flag is set to indicate that this is a special Data message used to inform the application of an error detected by the local node.
- The ECI application flag is set to indicate that the received chain has now terminated.
- The first four bytes of the buffer element hold the SNA sense codes detected by the local node that caused the termination.

This mechanism allows:

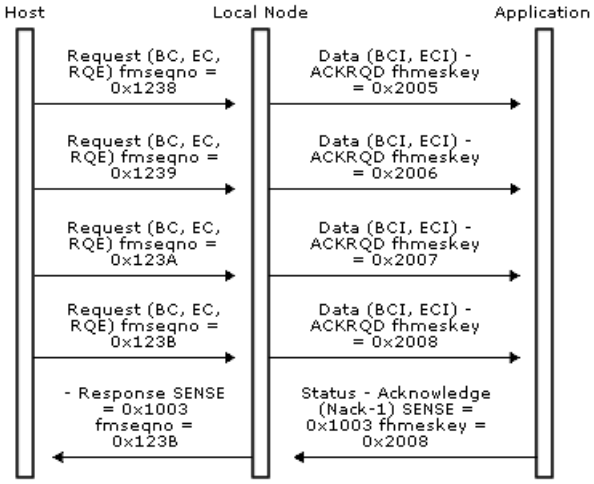
- The application to reject any previously-received **Data** messages.
- The local node to inform the application of any errors it detects in received requests.
- The local node to send negative responses in the correct order.

The following three figures illustrate the outbound data confirmation and rejection protocols between the local node and the application and how those protocols relate to the underlying SNA protocols.

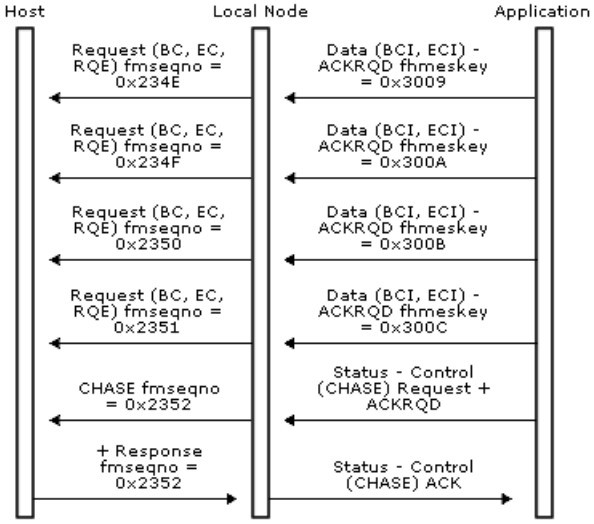
In the first illustration, the host sends a definite response chain to get the application to confirm receipt of the RQD request and all previously sent RQE chains.



In the following illustration, a **Status-Acknowledge(Nack-1)** from the application rejects the last chain and confirms receipt of all previously sent data chains.



In the following illustration, the host sends a CHASE request to get the application to confirm receipt of the CHASE and all previously sent chains.



Shutdown and Quiesce

Both shutdown and quiesce protocols involve a half-session entering a quiesced state, in which it cannot send any more normal flow requests, but must continue to receive and respond to requests from its session partner. The essential differences are that shutdown can only be initiated by the host and only requires that the secondary quiesce as soon as is convenient (usually at the end of a bracket); quiesce can be initiated by both the host and the application and requires that the recipient quiesce at the end of the chain.

If the application has been quiesced but still attempts to send inbound Data messages, they will be rejected with Status-Acknowledge(Nack-2) messages. The application can, however, continue to generate status messages.

This section contains:

- Shutdown
- Quiesce

Shutdown

The shutdown protocol provides a means for the host application to stop the application from sending any further normal flow requests. This protocol is used when the host application wishes to end the session in an orderly manner and is only available for sessions using FM profile 3 or 4.

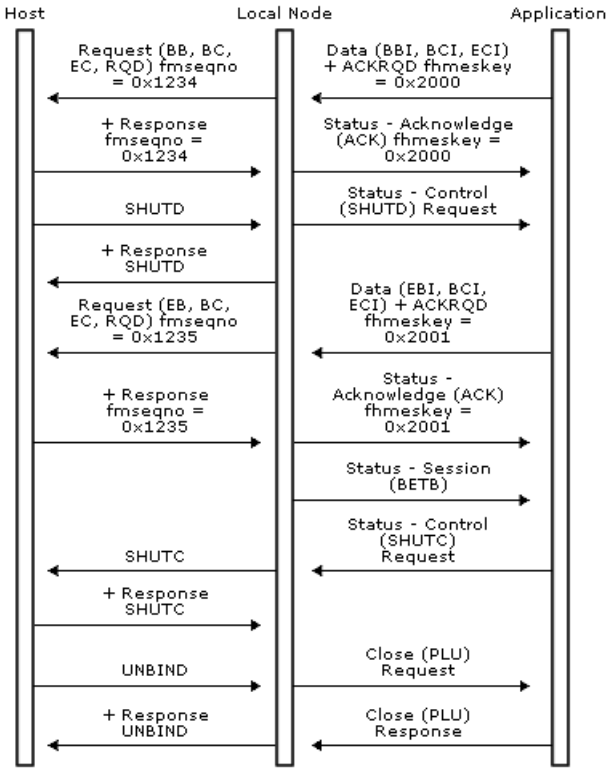
If the local node receives a SHUTD request from the host, it issues a Status-Control(SHUTD) Request (without ACKRQD) to request the application to enter a quiesced state at a convenient time. The application determines what is convenient — for example, it could be after a Status-Session(BETB) has been received.

When the application decides it is ready to quiesce, it should issue a Status-Control(SHUTC) Request (again without ACKRQD) to indicate this transition. The local node will notify the host of this change by sending a SHUTC request. The host can continue sending normal flow outbound requests and can subsequently take one of the following actions:

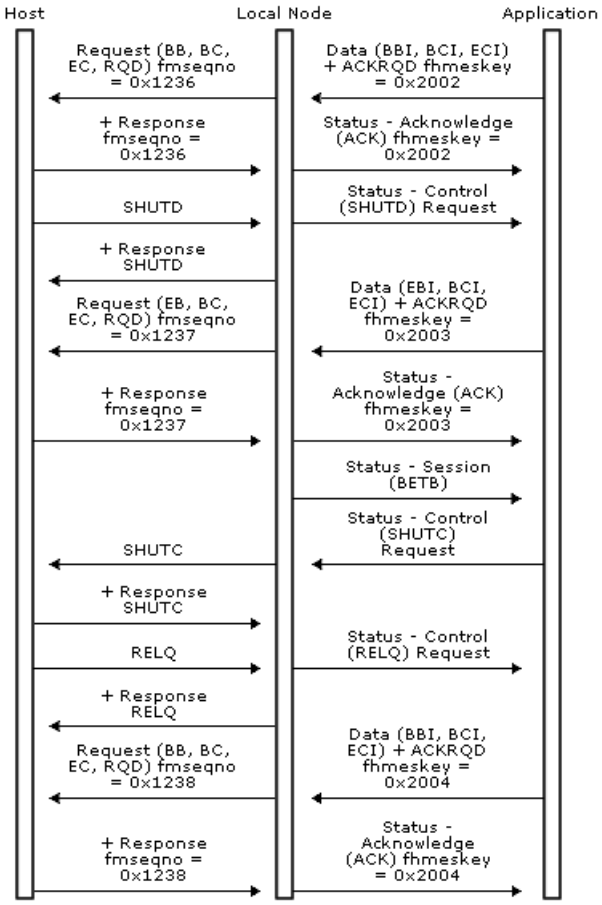
- The host terminates the PLU session by sending an UNBIND request. The local node closes the PLU connection by sending a Close(PLU) Request to the application. The SSCP session remains active.
- The host abandons the shutdown procedure by sending an RELQ request. The local node sends a Status-Control(RELQ) Request (with ACKRQD) to the application to indicate that it can now resume sending on the PLU session. RELQ is only supported on sessions using FM profile 4.
- The host resets the session by sending CLEAR (TS profile 3 or 4); one of the effects of this is to release the quiesced state (see Recovery).

The following two figures illustrate the shutdown protocols between the local node and the application and how those protocols relate to the underlying SNA protocols.

In the following illustration, the host sends SHUTD while the application is sending in the in-bracket state; the application completes the bracket, sends Status-Control(SHUTC) Request, and the host terminates the PLU session by sending UNBIND. The local node closes the PLU connection.



In the following illustration, the host sends SHUTD while the application is sending in the in-bracket state; the application completes the bracket, sends **Status-Control(SHUTC) Request**, and then the host releases the application from the quiesced state by sending RELQ. The local node sends a **Status-Control(RELQ) Request** to the application, which initiates a new bracket.


Host
Local Node
Application
Request (BB, BC, EC, RQD) fmseqno = 0x1236
Data (BBI, BCI, ECI) + ACKRQD fhmeskey = 0x2002
+ Response fmseqno = 0x1236
Status - Acknowledge (ACK) fhmeskey = 0x2002
SHUTD
Status - Control (SHUTD) Request
+ Response SHUTD
Request (EB, BC, EC, RQD) fmseqno = 0x1237
Data (EBI, BCI, ECI) + ACKRQD fhmeskey = 0x2003
+ Response fmseqno = 0x1237
Status - Acknowledge (ACK) fhmeskey = 0x2003
Status - Session (BETB)
SHUTC
Status - Control (SHUTC) Request
+ Response SHUTC
RELQ
Status - Control (RELQ) Request
+ Response RELQ
Request (BB, BC, EC, RQD) fmseqno = 0x1238
Data (BBI, BCI, ECI) + ACKRQD fhmeskey = 0x2004
+ Response fmseqno = 0x1238
Status - Acknowledge (ACK) fhmeskey = 0x2004

Quiesce

The quiesce protocol is only supported on sessions using FM profile 4. The quiesce protocol can be initiated by either half-session.

When an application wishes to quiesce its partner half-session in the host, it sends a Status-Control(QEC) Request to the local node. The node generates a QEC request to the host, which asks the host to quiesce after completing the current outbound chain.

If the host quiesces, it sends a QC request, which the local node presents to the application as a Status-Control(QC) Request (with ACKRQD). The host remains in a quiesced state until the application sends a Status-Control(RELQ) Request. The local node sends the RELQ request to the host, and the host resumes communications on the PLU session.

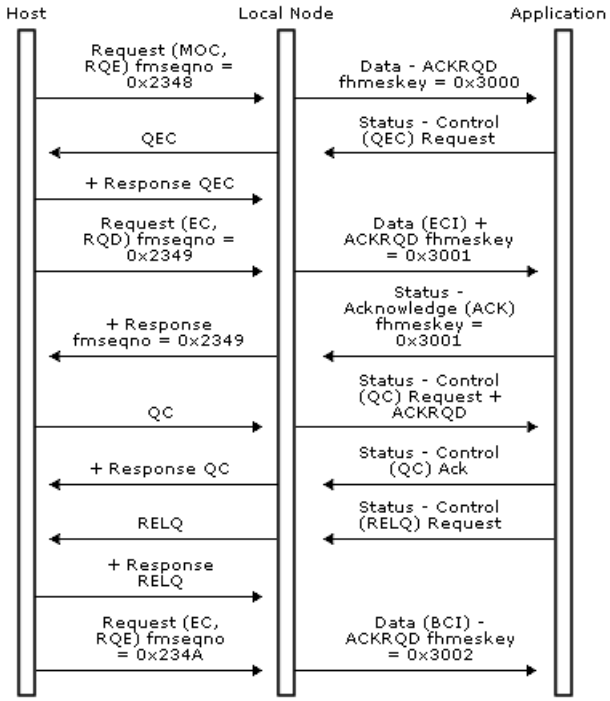
If the attempt to quiesce the host fails, the host responds with a negative QEC response, which the local node presents to the application as a Status-Control(QEC) Negative-Acknowledge-1.

Conversely, a Status-Control(QEC) Request (without ACKRQD) is presented to the application if a QEC request is received from the host. In this direction QEC cannot be rejected; the local node will always force the application to quiesce after presenting it with a Status-Control(QEC) Request by rejecting further attempts to send inbound data. When the application has quiesced, it should send a Status-Control(QC) Request to the local node, which sends a QC request to the host. The application can subsequently be released by an RELQ request from the host, which the local node presents to the application as a Status-Control(RELQ) Request.

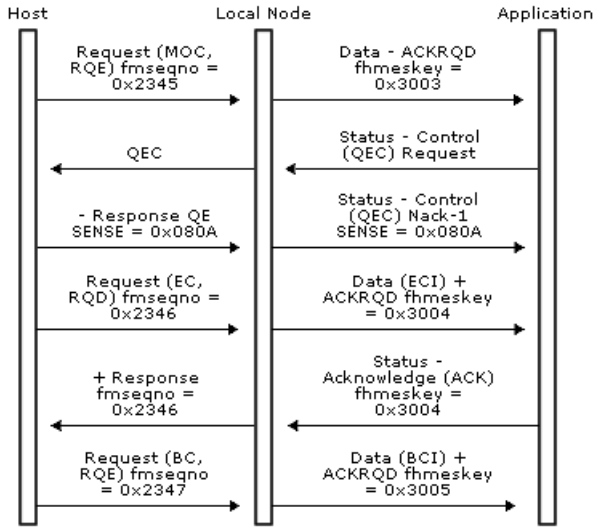
The receipt of a CLEAR or UNBIND–BIND sequence (Close(PLU)–Open(PLU)) causes the quiesced state to be released.

The following three figures illustrate the quiesce protocols between the local node and the application and how those protocols relate to the underlying SNA protocols.

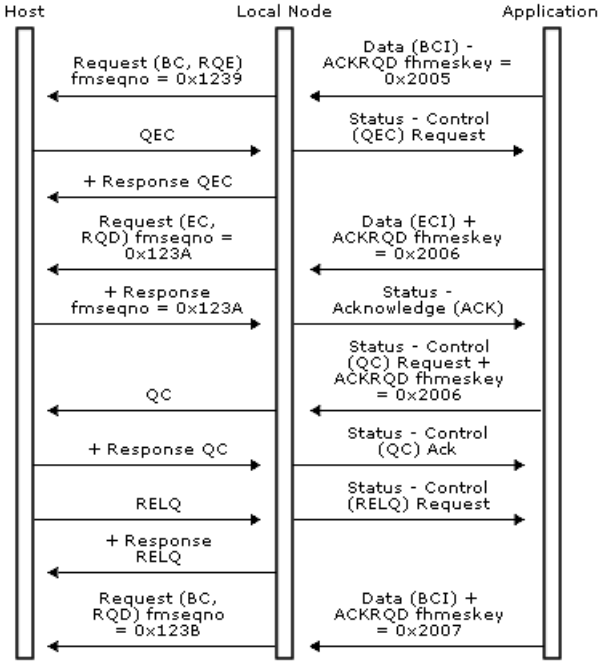
In the first illustration, the application quiesces the host and then releases the quiesce.



In the following illustration, the application attempts to quiesce the host, but the host rejects the quiesce and continues with the next chain.



In the following illustration, the host sends QEC while the application is sending a chain; the application completes the chain and sends a **Status-Control(QC) Request**. The host releases the quiesce by sending RELQ, and the local node sends a **Status-Control(RELQ) Request** to the application, which then initiates a new chain.



Recovery

This section covers a variety of issues pertaining to error recovery.

This section contains:

Application CANCEL

One of the parameters on the Open(PLU) OK Response, which the application sends to the local node, specifies whether the application will generate CANCEL (or EC) to terminate an inbound chain that has received a negative response. If this option is not selected, the local node will generate a CANCEL request when it receives a negative response from the host to an incomplete chain.

Direction After Receiving a Negative Response

Within the local node, error recovery for a half-duplex application (as specified by byte 7 bit 2 of the BIND) is assumed to be the responsibility of the host. However, the application must be aware of the fact that an error recovery state has been entered to obey the direction protocol.

When an application using half-duplex protocols (flip-flop or contention) receives a negative response to an inbound chain that it sent that does not refer to a race, it must assume receive state. The sense codes used for race conditions that do not require the transition to receive state are:

0x080B	Bracket race error
0x081B	Receiver in transmit mode

Direction After Sending a Negative Response

When an application using half-duplex flip-flop protocol sends a negative response to an outbound chain (or sends a Status-Acknowledge (Ack) to a DATAFMI message with SDI set) that does not refer to a race, the application must assume an error recovery pending state. The sense codes used for race conditions that do not require the transition to error recovery pending state are:

0x080B	Bracket race error
0x0813	Bracket bid reject (no RTR forthcoming)
0x0814	Bracket bid reject (RTR forthcoming)
0x081B	Receiver in transmit mode

The application must therefore examine the sense code on an SDI message to detect such races.

Error recovery pending state differs from receive state only in one respect: The application can convey sense information to the host using Status-Control(LUSTAT) — see LUSTATs. The LUSTAT must not have the CD or EB flags set (the host already has direction, and the bracket must not be terminated prematurely by the application). Host Integration Server or SNA Server also allow the FMI application to send Status-Control(LUSTAT) in receive state (see LUSTATs).

An application using the half-duplex contention protocol does not have an error recovery pending state, and must enter contention state whenever it sends a negative response.

Note that if the chain is canceled by the host with CD on the CANCEL, the application must assume send state.

Critical Failure

When an application makes a protocol error in sending data, the local node rejects the data using a Status-Acknowledge(Nack-2) with a sense code indicating the reason for failure. This message has a critical failure flag that indicates whether the local node has marked the session as unrecoverable. The sense codes are listed in FMI Status, Error, and Sense Codes.

If the error is noncritical, the application can proceed as if the message that caused the error had not been sent. If the error is critical, then the local node issues a Close(PLU) Request to the application (providing that the PLU connection is open), which means that the application cannot communicate on the PLU-SLU session until an UNBIND–BIND sequence is received from the host. The local node also sends a TERM-SELF request to the host to elicit an UNBIND; therefore, the application does not need to issue a LOGOFF request on the SSCP session.

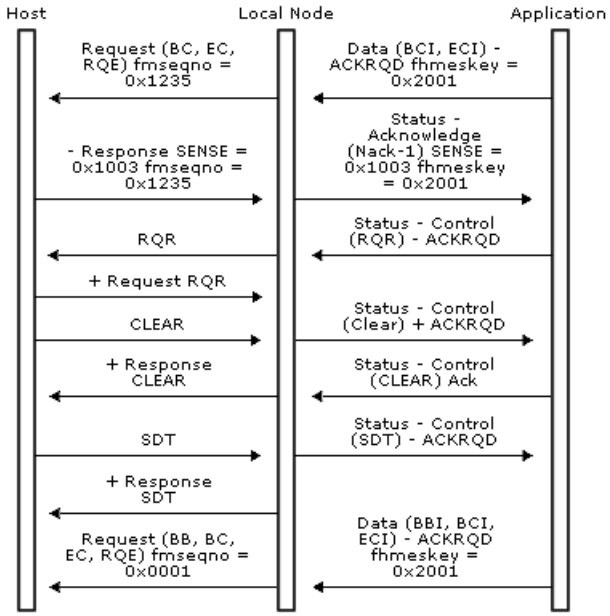
RQR and CLEAR

An application using TS profile 4 can request the session to be recovered by sending **Status-Control(RQR)**; the local node presents this to the host as an RQR request. Note that, if the application has received a critical Status-Acknowledge(Nack-2), this option cannot be taken because the local node will send a Close(PLU) Request immediately following the **Status-Acknowledge(Nack-2)** to the application, and the PLU connection will no longer be valid. The RQR message requests the host to reset the session by sending a CLEAR request (see the following figure).

The receipt of CLEAR causes the application to reset its session state to that following the BIND (that is, the Open(PLU)).

Another way for the application to deal with error conditions is to ask for an UNBIND by sending Status-Control(RSHUTD) (see Application-Initiated Termination). Note that this may not require the host to supply a new BIND, depending on the host configuration. A new SSCP request may be required (such as LOGON).

In the following illustration, the application requests recovery by issuing Status-Control(RQR); the host sends CLEAR, and the application must reset its session to state that it was in following the BIND (Open(PLU)). In this case, this means that the application is now between brackets and awaiting SDT.



STSN

STSN (set and test sequence numbers) is used on sessions with TS profile 4 for applications to maintain transaction processing sequence numbers between sessions. This allows both partners on the session to discover the amount of data lost after a CLEAR or UNBIND–BIND sequence.

The STSN message is the only one that can reset such transaction processing sequence numbers; BIND, UNBIND and CLEAR do not affect them.

If the application wishes to maintain such transaction numbers, it must specify the APPLTRAN option in the Open(PLU) OK Response. The host can send STSN after a BIND or CLEAR before sending SDT to set and/or test the application's transaction numbers. The local node resets its internal session sequence numbers to zero on receipt of BIND or CLEAR. When the local node receives an STSN specifying SET (or SET and TEST) for one half-session, it resets the corresponding internal session sequence number.

Unless both half-session actions are ignore (the action byte is 0x00), the STSN request is passed to the application (provided that it specified APPLTRAN), with the action byte and the two sequence numbers from the request, as a Status-Control(STSN) (see Status-Resource). The application must examine the action byte to determine whether the action is ignore, set, test, or set and test. The application must send a positive response (Status-Control(STSN) Acknowledge) to the STSN, with the sensed sequence numbers if required (sense or set and test). The local node is responsible for generating the correct result code for the STSN RSP.

Note that the application should perform the sense part of STSN first (by examining bits 0 and 2 of the action byte for the secondary-to-primary flow and primary-to-secondary flow respectively). The set part of the STSN is then performed (by examining bits 1 and 3 of the action byte).

The application should increment its transaction numbers when sending and receiving normal flow RUs from the host (note that Status-Control messages corresponding to normal flow DFC requests cause the transaction numbers to be incremented). The sequence number is reported on DATAFMI messages and Status-Acknowledge messages. The application should be aware that, if a message from the host fails receive checks (and is converted to an SDI message), SNAP-2.1 will purge the remainder of the chain from the host, and the application may miss some sequence numbers. Therefore, the application should reset its primary-to-secondary transaction number from the next outbound data after processing an SDI message.

Note that the second application flag byte is not valid for Status-Control(STSN); it is used for the STSN control byte.

Link Service Failure

When the server running a link service fails, the local node is informed of this; it treats the problem as a link outage with outage code 0x0D. This is reported to any active 3270 emulation sessions as a communications check code (–+z_nnn). The local node will attempt periodically to reconnect to the link service.

Local Node Failure

If the local node fails, applications are informed of this by the path error return code from the DMOD on the sbpurcvx call, or from the routing procedure. All connections that use the destination locality value for which the path error is reported are closed. The application must:

- Tidy up resources related to the closed connections, including resetting presentation spaces and displaying a communications check code
  (–+z_nnn) on the status line.
- Attempt to reestablish connection with a local node by reinitiating the resource location process.

Client Failure

If the client computer fails, the local node terminates the application's PLU-SLU session (if it is active) by sending TERM-SELF. The SSCP and PLU connections are both marked as closed and cannot be reused without being reopened. Internally, the local node treats such a failure in the same way as the receipt of a Close(SSCP) Request from the application.

Application-Initiated Termination

An application on a session with FM profile 3 or 4 can request termination of the PLU session. It should only do so if it has previously ensured that it is in a state where the PLU session can be terminated, that is, between-chain and between-bracket. Terminating the PLU session does not affect the state of the SSCP session.

Note that an application can issue a character coded or field formatted LOGOFF command on the SSCP session or send a Close(PLU) Request to get the local node to send TERM-SELF on the application's behalf. All of these will elicit an UNBIND, either immediately or after session tidy-up in the host.

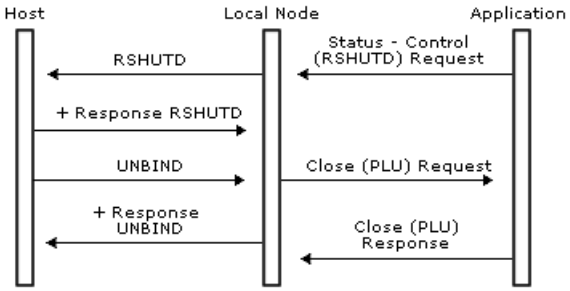
The application requests termination of the PLU session by sending a Status-Control(RSHUTD) Request to the local node, which generates an SNA RSHUTD request to the host.

After sending the Status-Control(RSHUTD) Request, the application must remain capable of accepting and responding to all outbound data it receives. The application can now expect one of two messages, depending on whether the state of the PLU session allows it to be terminated and whether the host wishes to terminate the PLU session:

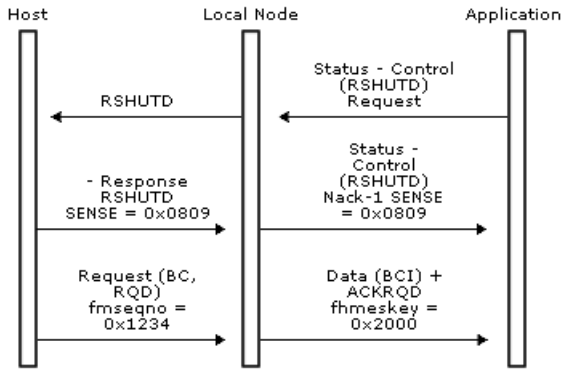
- If the state of the PLU session allows it to be terminated, and the host wishes to terminate the PLU session, the host generates a positive response to the RSHUTD request, which can be followed by an UNBIND request. The local node closes the PLU connection; see Closing the PLU Connection.
- If the state of the PLU session does not allow it to be terminated (for example, if the session is in-bracket), or the host does not want to terminate the PLU session at this time, the host generates a negative response to the RSHUTD request, which the local node presents to the application as a Status-Control(RSHUTD) Negative-Acknowledge-1 carrying the sense codes supplied on the negative response. This indicates that the request to terminate the PLU session has been rejected by the host, and communication on the PLU session continues unaffected.

The following two figures illustrate the application-initiated termination protocol between the local node and the application and how this protocol relates to the underlying SNA protocols.

In the first illustration, the application requests termination of the PLU session, and the host sends UNBIND. The local node closes the PLU connection.



In the following illustration, the application requests termination of the PLU session, but the session is not in an appropriate state. The host sends a negative response to the RSHUTD request, which the local node presents as **Status-Control(RSHUTD) Negative-Acknowledge-1**. Communication continues on the PLU session.



LUSTATs

The DFC LUSTAT message is used within SNA to convey four bytes of sense data to the other session partner. It can also be used simply to send an RH to the other session partner (for example, to open a bracket; see the figures in Bracket Initiation). The message flows on the normal flow and so is subject to direction restrictions; however, it can be sent (without EB or CD) on a half-duplex flip-flop session that is in error recovery pending state (see Recovery).

The local node allows the application to send Status-Control(LUSTAT) Request messages at any time that data traffic is active, except while sending data in chain. If the application is in a receiving state (using half-duplex protocol), the LUSTAT is queued up and used to provide the sense codes, which are filled into the next outbound request, and the SDI flag is set. The application can therefore present the sense codes for an error state without waiting for the next outbound data if required.

The first byte of sense data must be 0x08 to generate a DATAFMI message with SDI (to be converted to a negative response). Other LUSTATs are left queued on the session until they can be sent.

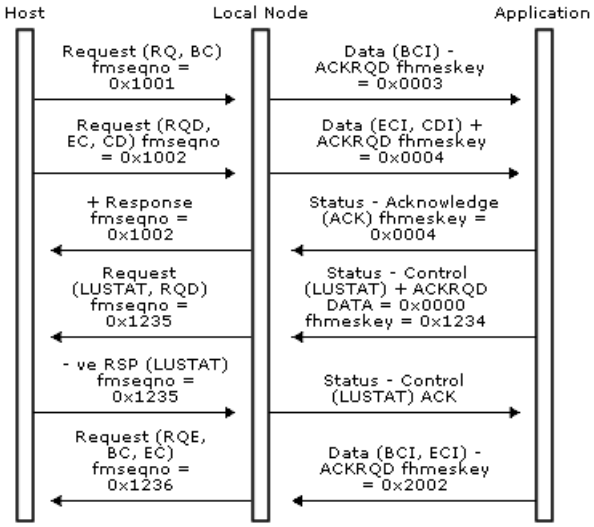
If multiple Status-Control(LUSTAT) messages are sent by the application while in a receive state, the local node will queue them all. When outbound data has been delivered to the application with SDI set, as indicated above, and the application has converted it to a Status-Acknowledge(Ack), the local node will send the negative response and the remaining LUSTATs (which can now flow since the half-duplex flip-flop state is error recovery pending).

If the application intends to send multiple Status-Control(LUSTAT) messages to the host, it is possible that the host will attempt to initiate recovery before the last LUSTAT has been sent; in this case, the error recovery chain will be rejected by the next LUSTAT.

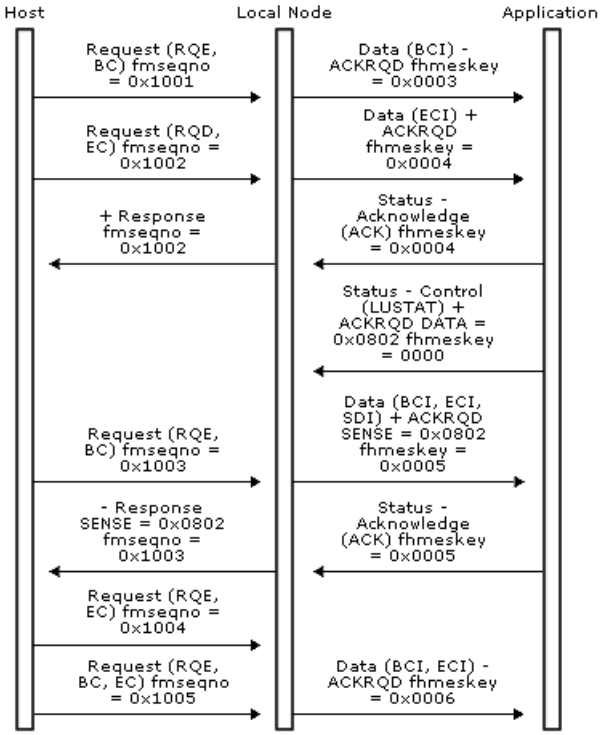
Note that the application can send Status-Control(LUSTAT) Request with or without ACKRQD; the local node will map these to RQD and RQE LUSTATs respectively.

The following three figures illustrate the use of Status-Control(LUSTAT) messages by an application using the half-duplex flip-flop mode.

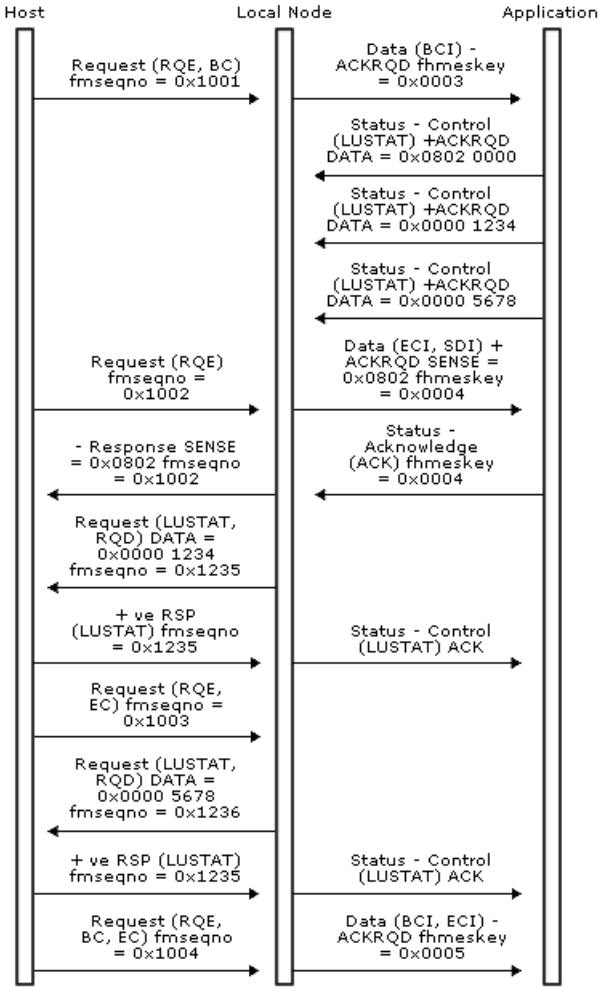
In the first illustration, the application issues Status-Control(LUSTAT) when it has direction.



In the following illustration, the application sends **Status-Control(LUSTAT)** request when receiving data between chain. Next outbound data is delivered with SDI set which gets converted to negative RSP.



In the following illustration, the application sends several **Status-Control(LUSTAT)** requests when receiving data in chain. Next outbound data is delivered with SDI set which gets converted to negative response. Subsequent LUSTATs are sent to host.



Response Time Monitor Data

For a 3270 display application, the local node maintains statistics on host response times — the time it takes the host to respond after the 3270 user presses ENTER or an AID key to send data to the host. These statistics can then be sent to the host for analysis.

The Status-RTM message, sent by the local node to the application, informs the application of the Response Time Monitor parameters specified by the host (see RTM Parameters for more information). These parameters specify whether RTM data is to be collected, whether the application is permitted to display RTM statistics locally, the time boundaries by which response times are to be grouped, and the definition of response time. The time can be measured until the first character of the host response reaches the screen, until the keyboard is unlocked, or until the user can send further data (CD or EB received by the application).

If the host specifies that response times are to be measured for this session, the application is responsible for measuring response times and for reporting them to the local node. This involves:

- Starting a timer when the user presses the ENTER key or an AID key to send data to the host.
- Stopping the timer when the host's response to the inbound data is received, as defined by the RTM definition specified on the Status-RTM message.
- Reporting the response time to the host on the Status-Acknowledge(Ack) message, which acknowledges the host's response. One of the fields on this message specifies the last response time measured by the application, or specifies that no response time is to be reported.
- Optionally displaying the most recent response time as a last transaction time indicator (LTTI) on the 3270 emulation status line.

If the application wishes to provide a local display of RTM data, it is responsible for maintaining its own response time statistics. It should use the same definition and boundaries as those specified on the Status-RTM message to ensure that the local data matches the data sent to the host by the local node. Note that the **Status-RTM** message can indicate that a local display of response times is not permitted; in this case, the application should not display either the response times or the LTTI.

Data Flow

The following topics describe data flows between the application and the local node.

This section contains:

Outbound Data

This section describes the outbound data flows from the local node to the application. The overall structure of the protocols described applies to the SSCP and PLU connections, but certain features (such as the use of delayed request mode) are only applicable to the PLU connection.

The local node presents data originating at the host to the application on different connections, depending on the SNA session on which the data flows, as follows:

- FMD NS (session services) data and FMD data originating at the host SSCP and directed to the Microsoft® Host Integration Server or Microsoft® SNA Server LU is sent to the application on the SSCP connection.
- FMD data originating at the host PLU and directed to an SNA server LU is sent to the application on the PLU connection.

For all connections, only FMD requests are presented to the application as Data messages (that is, with message-type = DATAFMI). DFC and session control requests are used to generate **Status-Control** messages (see Status-Control Message).

The local node performs the data flow control state changes required by the RH indicators in the request, before sending a Data message to the application.

The SNA request TH (transmission header) and RH indicators are not available to the application on outbound Data messages. Instead, the local node provides application flags in the Data message header that reflect the settings of a subset of the RH indicators, but are interpreted by the local node to shield the application from the more obscure aspects of chaining and bracket usage. See Application Flags for a description of the available flags and the way in which the local node uses them on outbound data.

For outbound data, the first byte is RU[0] for standard FMI, and TH[0] for the LUA variant of FMI.

All Data messages from the local node to an application contain a message key. The local node maintains a unique message key sequence for each outbound data flow to an application. When the local node sends a Data message to an application on a particular connection, it places the next message key in the outbound sequence into the message header, sets the application flags, and sends the message to the application. This means that the message key uniquely identifies a Data message on a particular connection between the local node and the application. Note that the local node also places message keys on outbound Status-Control Request messages.

The acknowledgment protocol enforced by SNA server reflects the chain response protocol and request mode in use on the SNA session, as follows:

- Outbound RQD requests generate Data messages with ACKRQD set in the message header.
- Outbound RQE requests generate Data messages without ACKRQD set.
- Outbound RQN requests generate Data messages without ACKRQD set.
- If the session uses primary immediate request mode, a Data message with ACKRQD set must be acknowledged by the application before further Data messages will be received.
- If the session uses primary delayed request mode, a Data message with ACKRQD set need not be immediately acknowledged by the application; Data messages will continue to be received.

Note that Host Integration Server or SNA Server enforce the equivalent of immediate response mode for the outbound data acknowledgment protocol for all connections. That is, the application must send acknowledgments in order.

If the local node sets the ACKRQD field in the message header of a Data message to the application, it indicates that an acknowledgment to this Data message is required. The application acknowledges an outbound Data message by sending a Status-Acknowledge message to the local node on the same connection, which contains the same message key and sequence number fields as the Data message.

On receipt of a Status-Acknowledge(Ack), the local node correlates the message key with outstanding outbound messages and generates an SNA positive response to the appropriate SNA request.

The application should use the Status-Acknowledge(Nack-1) message as a negative acknowledgment. On receipt of a Status-Acknowledge(Nack-1), the local node correlates the message with outstanding outbound messages and generates an SNA negative response plus sense data to the appropriate SNA request. The application supplies the sense data that should accompany the negative response as part of the Status-Acknowledge(Nack-1) message and must include the same message key, application flags, and sequence number fields as the Data message to which this is a negative acknowledgment.

Status-Control messages caused by expedited-flow requests can be sent at any time and do not affect the sending of positive or negative acknowledgment to normal flow outbound Data messages. The fact that they can occur between an outbound Data message and the matching Status-Acknowledge message is purely coincidental. See Status-Control Message for details of which

Status-Control messages correspond to SNA requests.

If errors are detected in the format of a normal flow request from the host or the request is inappropriate for the state of the session, the local node generates an error Data message for the application with the following characteristics:

- The SDI and ECI application flags are set.
- The sense code associated with the error occupies the first four bytes of the Data message (see Status-Control Message).
- ACKRQD is set.

The application should return a Status-Acknowledge(Ack), and the local node generates a negative response carrying the sense code appropriate to the detected error. This mechanism:

- Informs the application of the detected error.
- Allows the application to respond to any previously-received data before the local node sends the negative response to this Data message.

On sessions where the application is receiving a series of RQE chains, the local node will be retaining correlation information for each chain (in case the application wishes to send negative responses to any of the chains). If the local node runs out of correlation table entries, it will attempt to allocate more entries and (if this fails) will be forced to terminate sessions. To prevent this, the application should provide **Status-Acknowledge(Ack)** messages for RQE data that it does not wish to reject in this case; a response after five consecutive RQE chains should be sufficient. Such messages are referred to as courtesy acknowledgements and do not give rise to a response to the host, but merely free internal correlation data.

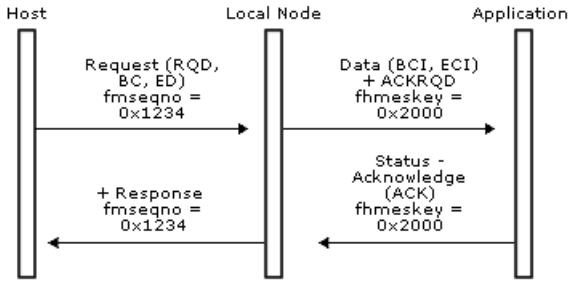
The following six figures illustrate the data acknowledgment protocol enforced between the local node and the application, and show the effects of the application generating positive and negative Status-Acknowledge messages.

The figures show:

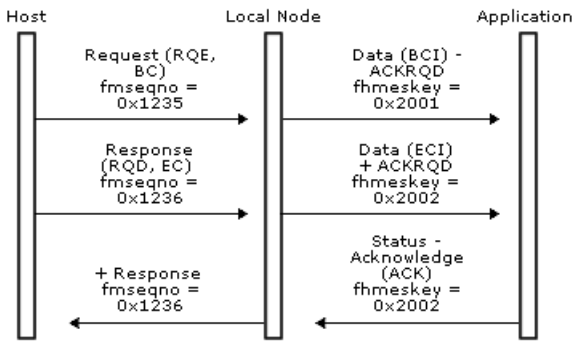
- The relevant RH flags in SNA requests/responses.
- The sequence number of SNA requests/responses.
- Any sense data (shown as "SENSE=...") on SNA requests/responses and Status-Acknowledge messages.
- The ACKRQD field in Data messages.
- The message key field in Data messages.

For simplicity, all messages are assumed to be FM data flowing on the same PLU session.

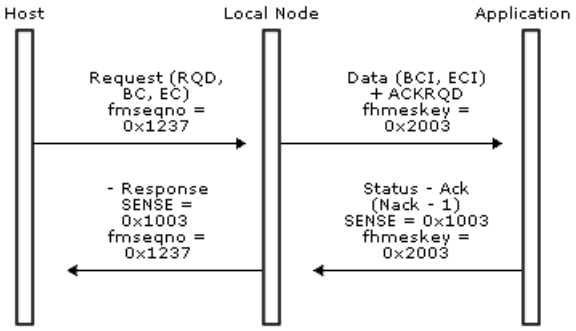
In the following illustration, the application accepts a Data message corresponding to a definite-response RU.



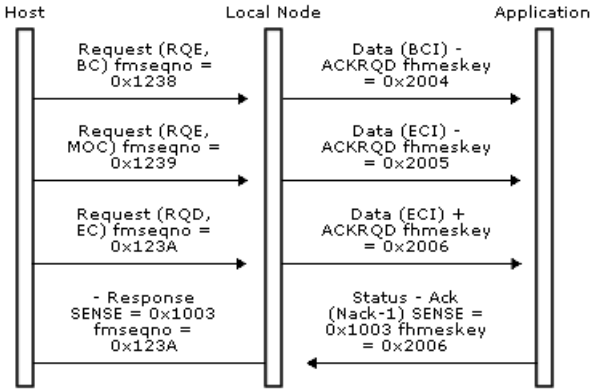
In the following illustration, the application accepts a **Data** message corresponding to a multi-RU definite-response chain.



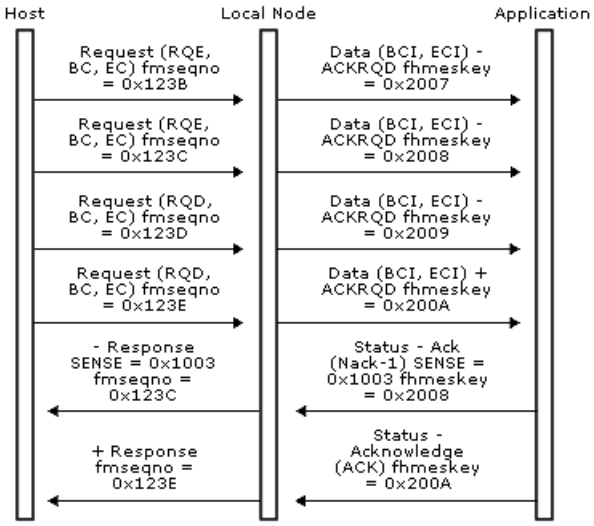
In the following illustration, the application rejects a **Data** message corresponding to a definite-response chain.



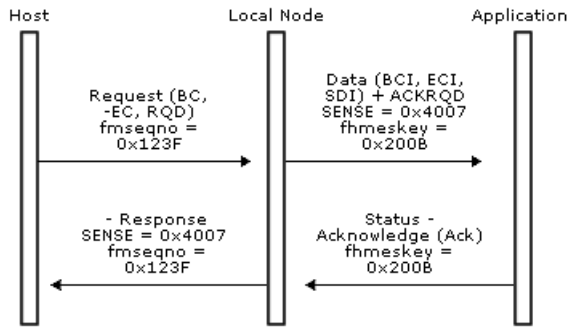
In the following illustration, the application rejects a **Data** message corresponding to a multi-RU definite-response chain.



In the following illustration, the local node enforces immediate response mode; that is, responses must be sent in sequence. The application rejects the second exception-response chain and accepts the definite-response chain, which implies acceptance of the third exception-response chain.



In the following illustration, the local node detects a chaining error (RQD but not EC) in data destined for the application (this example requires the receive check 0x4007 to be in force — see Opening the SSCP Connection).



Inbound Data

This section describes inbound data flows from the application to the local node. The overall structure of the protocols described applies to the SSCP and PLU connections, but more complex aspects (such as the use of delayed request mode) are only applicable to the PLU connection.

The application can send inbound data on any of the three connections, as follows:

- FMD NS (session services) and FMD character-coded data intended for the host SSCP should be sent to the local node on the SSCP connection.
- FMD data intended for the host PLU should be sent to the local node on the PLU connection.

The application cannot use Data messages to send DFC or session control request messages to the host. Instead it must use **Status-Control** messages (see Status-Control Message for further details).

For all three connections, the application must fill in certain key fields in the Data message's header. In particular it must:

- Set the message-type to DATAFMI.
- Allocate a new message key for inbound Data messages on this connection.
- Set the ACKRQD field if required (see below).
- Set the application flags (see Application Flags).

The **nxtqptr**, **hdreptr** and **numelts** fields in the message header, and the **elteptr** and **startd** fields in the message elements are set up by the Host Integration Server or SNA Server buffer management routines (see The DL-BASE/DMOD Interface). The application is responsible for setting the **endd** field.

Where the application does not have access to these routines (for example, where the operating environment does not support intertask procedure calls and shared memory), all the fields in the header must be set by the application.

The TH and RH indicators are not available to the application on inbound Data messages. The application should set the appropriate application flags in the message header to control chaining, direction, and so on — see Application Flags for a description of the available application flags for inbound data and later topics in this section for a description of how the flags are used to control inbound data flows.

For inbound data, the first byte is RU[0] for standard FMI.

The message key supplied by the application in the inbound Data message header is used by the local node to indicate which Data message on this connection an outbound Status-Acknowledge refers to. The application should maintain a unique message key sequence for the inbound data flow on each connection it has with the local node, so that the application can use the message key to correlate inbound Data messages and outbound Status-Acknowledge messages on the connection. Note that the application must also provide a message key on Status-Control Request messages to differentiate between multiple RQE LUSTAT messages.

The inbound data acknowledgment protocol reflects the secondary chain response protocol and request mode in use on the session, as follows:

- Inbound Data messages with ACKRQD set in the header generate RQD requests.
- Inbound Data messages without ACKRQD set in the header generate RQE or RQN requests depending on the chain response protocol.
- The application should only set ACKRQD on Data messages that have the ECI (end chain indicator) application flag set.
- If the session specifies that the secondary uses immediate request mode, the application can still send further Data messages after sending data with ACKRQD set, even though it has not received a Status-Acknowledge message for that Data message. The messages are queued within the local node and are progressively sent as positive responses are received.
- If the session specifies that the secondary uses delayed request mode, then after sending a Data message with ACKRQD set, the application can continue to send Data messages.

If the application sets the ACKRQD field in the message header of a **Data** message, it indicates that it requires an acknowledgment to this **Data** message. The local node acknowledges an inbound **Data** message by sending a **Status-Acknowledge** message to the application on the same connection and using the same message key as the **Data** message (see the first figure at the end of this topic).

The local node processes inbound Data messages from the application through its internal state machines, assigns the correct SNA sequence number or an identifier for this flow, and sends the data in a request to the host. The chain-response type of the

request depends on whether ACKRQD was set in the Data message and the session parameters.

The local node maps a positive response from the host to a Status-Acknowledge(Ack) to the application. The application can use the message key in the Status-Acknowledge to correlate the acknowledgment with the original Data message. Therefore, receipt of a Status-Acknowledge(Ack) for a particular Data message implies that the local node has received a positive SNA response from the host to the inbound SNA request (see the second figure at the end of this topic).

Note that responses are absorbed on the SSCP-PU session.

Note that outbound Status-Acknowledge(Ack) messages contain application flags and a sequence number. The application flags reflect the RH indicators in the response. The sequence number is the SNA sequence number from the response, and provides a mechanism for applications using TS profile 4 to track the SNA secondary sequence number corresponding to a unit of work.

The local node maps a negative response from the host to a Status-Acknowledge(Nack-1) message to the application. The application can use the message key in the Status-Acknowledge to correlate the negative acknowledgment with the original Data message. The outbound Status-Acknowledge(Nack-1) message contains the SNA sense codes and sequence number from the negative response (see the third and fourth figures at the end of this topic).

If the local node detects an error in the format of an inbound Data message, or the Data message is not appropriate to the current state of the session, it sends a Status-Acknowledge(Nack-2) to the application containing an error code (see Error and Sense Codes for a list of error codes). The local node does not send a request to the host corresponding to the Data message in error and does not advance the SNA sequence number for the session. The application can use any message key in its next inbound Data message (assuming the error does not cause a critical failure).

An example of a serious chaining error, where the application sends a Data message with ACKRQD but without ECI in the application flags, is shown in the last figure at the end of this topic. Note that after detecting this particular error, the local node marks the application's connection as critically failed, closes the connection, and sends a TERM-SELF request to the SSCP to elicit an UNBIND (see Recovery).

Inbound Status-Control messages, which cause the generation of expedited-flow requests, can be sent at any time and do not affect the sending of a positive or negative acknowledgment to inbound Data messages. See Status-Control Message for details of which Status-Control messages correspond to SNA expedited-flow requests.

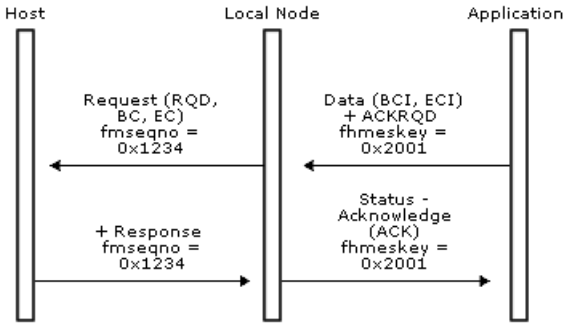
The following five figures illustrate examples of the inbound data acknowledgment protocols (and the underlying SNA protocols) for different chain-response types and secondary session request modes.

The figures show:

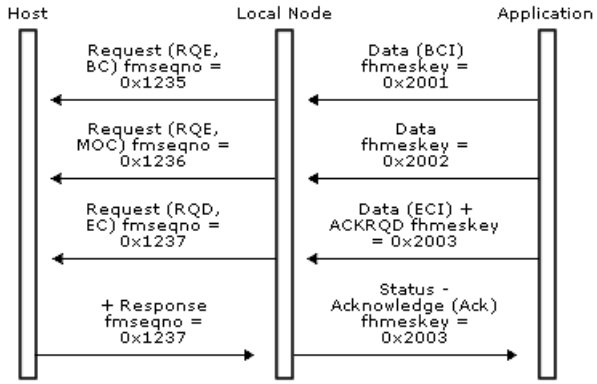
- The ACKRQD field on **Data** messages.
- The message key on Data messages.
- Any relevant application flags on Data messages.
- Error codes (shown as "ERROR=...") on Data messages.
- Relevant RH flags on SNA requests/responses.
- Sequence numbers on SNA requests/responses.
- Sense codes (shown as "SENSE=....") on SNA requests/responses.

For simplicity, all messages are assumed to be flowing on the same PLU session.

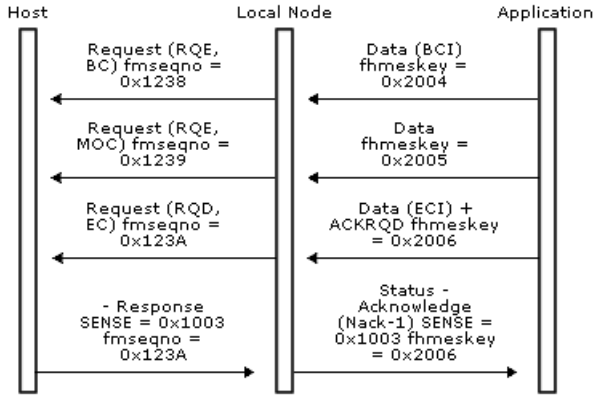
In the following illustration, the application successfully sends a Data message.



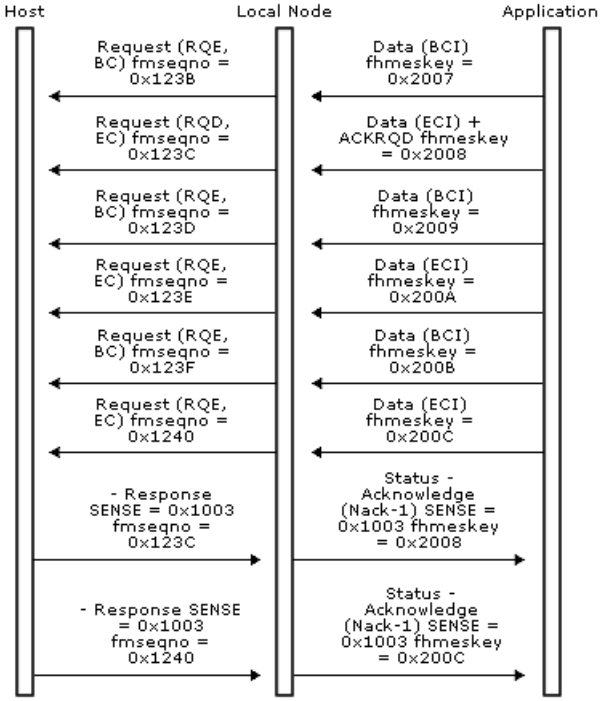
In the following illustration, the application successfully sends a chain of **Data** messages.



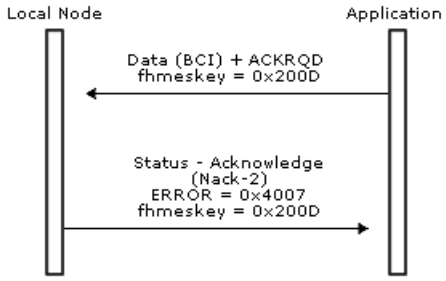
In the following illustration, the host rejects a chain of **Data** messages.



In the following illustration, the host rejects the first definite-response chain and rejects the third exception-response chain on a delayed request session. Note that the negative response to the third chain implies a positive response to the second chain.



In the following illustration, the local node detects the application's invalid use of ACKRQD without the ECI application flag on a **Data** message. Note that no data is sent to the host; however, since the error is critical, the local node will send a TERM-SELF message to the SSCP.


Local Node
Application
Data (BCI) + ACKRQD
fhmeskey = 0x200D
Status - Acknowledge
(Nack-2)
ERROR = 0x4007
fhmeskey = 0x200D

Inbound Data from LUA Applications

As shown in the previous topic, the local node performs certain checks on data supplied by a client application before sending it to the host and rejects it with a Status-Acknowledge(Nack-2) message if the checks fail; it does not return any acknowledgment to the application if the data passes the checks (although the host may do so later).

Where the client application is providing an LUA API, the design of the API may require that an LUA verb sending data inbound to the application does not complete until the local node has checked the data. Because of this, the local node will always respond to a client application that uses the LUA variant of the FMI, after it has completed its send checking of the inbound message; this allows the client application to complete processing of the LUA verb and return control to the LUA application program.

If the inbound message passed the local node's send checks and will be sent to the host, the local node sends a Status-Acknowledge(ACKLUA) message to the client application to indicate this; the client application can then complete the LUA verb processing with an OK return code. Note that the Status-Acknowledge(ACKLUA) message does not imply that the data was successfully sent to the host or that the host received it; it may later be followed by a Status-Acknowledge(Nack-1) message indicating that the host rejected the data.

If the inbound message fails the local node's send checking, a Status-Acknowledge(Nack-2) message will be returned as for non-LUA client applications. The client application can then report this to the LUA application program by a non-OK return code to the LUA verb that sent the message.

If the client application is providing an LUA API, it should therefore wait for either Status-Acknowledge(ACKLUA) or Status-Acknowledge(Nack-2) to determine whether to return an OK or error return code to the LUA send verb. If this dependence on the local node's send checks is not required, the client application can ignore the Status-Acknowledge(ACKLUA) message.

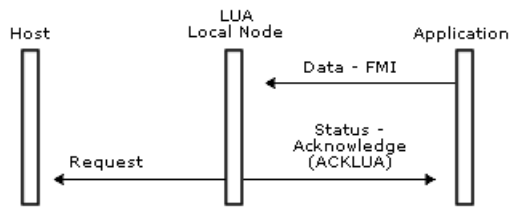
Note that there are certain race conditions in which the local node cannot complete its send checks before replying to the client application. In these cases, the local node returns a Status-Acknowledge(ACKLUA), but may subsequently send a Status-Acknowledge(Nack-2) if it detects an error during the remaining send checks. The client application may therefore receive a Status-Acknowledge(ACKLUA) followed by a Status-Acknowledge(Nack-2) for the same inbound message.

In the TH for the LUA variant of FMI, the expedited flow indicator (EFI), the destination-address field (DAF), and the origin-address field (OAF) are used. Other fields (including the sequence number field) are ignored. In the RH for the LUA variant of FMI, all fields except the queued-response indicator (QRI) and pacing indicator (PI) are used.

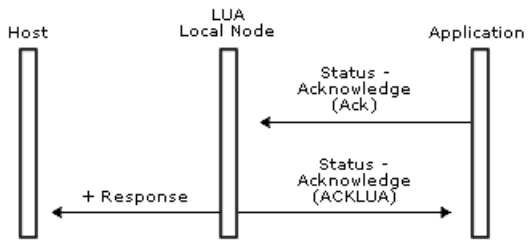
For inbound data, the first byte is TH[0] for the LUA variant of FMI.

The following three figures illustrate the Status-Acknowledge(ACKLUA) acknowledgment protocol for different messages that the application can send.

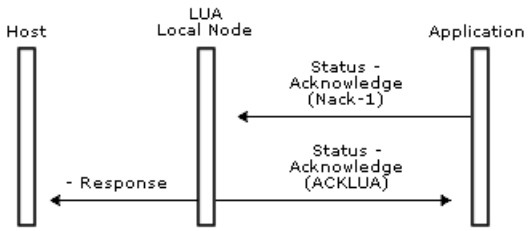
In the first illustration, the application sends a Data message that passes the local node's send checks.



In the following illustration, the application sends a **Status-Acknowledge(Ack)** message that passes the local node's send checks.



In the following illustration, the application sends a **Status-Acknowledge(Nack-1)** message that passes the local node's send checks.


Host
LUA
Local Node
Application
Status -
Acknowledge
(Nack-1)
Status -
Acknowledge
(ACKLUA)
- Response

Status Messages

The local node uses status messages to provide the application with information about the state of its sessions and to give the application control (in association with the host SSCP and PLU) over the progress of the session. The status messages are designed to allow the application to use all the protocols allowed in the FM and TS profiles supported by the Microsoft Host Integration Server or Microsoft SNA Server local node.

Most applications only need to use a subset of the available status messages — for example, a 3270 device emulator would not require the status messages used in quiesce protocols.

This section contains:

Status-Acknowledge Message

Status-Acknowledge messages are the basic mechanism used to enforce inbound and outbound data acknowledgment protocols on all three connections. See Outbound Data and Inbound Data for details of the use of **Status-Acknowledge** messages.

For a 3270 emulation application, Status-Acknowledge messages sent from the application to the local node (acknowledging outbound data from the host) can also carry information on host response times. This allows the local node to maintain response time statistics for sending to the host when required. See Response Time Monitor Data for details.

Status-Control Message

Status-Control messages provide access to session control and data flow control protocols on the PLU session using the PLU connection; they are not used on either of the other connections. **Status-Control** messages map directly to the equivalent SNA session control and data flow control RUs.

All Status-Control messages that correspond to SNA requests on the normal flow with the exception of LUSTAT-sent RQE, and Status-Control messages corresponding to CLEAR and STSN request on the expedited flow, have the ACKRQD (acknowledgment required) field set. Status-Control messages that correspond to SNA requests on the expedited flow (with the exception of CLEAR and STSN) do not have the ACKRQD field set by the local node. However, the application can set ACKRQD when sending these Status-Control messages. The last figure in this topic summarizes which Status-Control requests always have ACKRQD set.

If a Status-Control request has ACKRQD set in the message header, then the recipient must supply a Status-Control response (that is, Acknowledge, Negative-Acknowledge-1 or Negative-Acknowledge-2) before the sender sends further Data messages or further Status-Control requests on the flow; the sender can still send Status-Control responses, Status-Acknowledge, Status-Error, and Status-Resource messages on the flow. This applies to both normal and expedited flows and all request modes (including delayed-request mode). The message key received on the request must be returned on the response (this is to allow multiple RQE LUSTAT messages to be outstanding). The local node increments the message key on Status-Control requests and DATAFMI messages that it sends to the application on the PLU connection.

For the LUA variant of the FMI, the message key field is used in a different way, as follows:

- For inbound expedited flow requests, the local node sets the SNA sequence number to the value supplied by the application in the message key field. The application must ensure that this field is set to the correct sequence number.
- For inbound Status-Control responses, the local node sets the SNA sequence number to the value supplied by the application in the message key field. The application must ensure that this field is set to the sequence number of the request for which a response is being sent.

Except in the case of **Status-Control(LUSTAT)**, if a **Status-Control** request does not have ACKRQD set then the application should not reply, since a positive response has already been sent by the local node.

For example, if the application sends a Status-Control(QC) Request with ACKRQD set (corresponding to an SNA request on the normal flow), this blocks further data and Status-Control requests corresponding to the inbound normal flow until the Status-Control(QC) response is received. It does not block other messages on the normal flow, or messages on the expedited flow; for example, the application could still send Status-Control(SIGNAL).

The receipt of the Status-Control response implies an acknowledgment to all outstanding messages (including Data messages) on the flow.

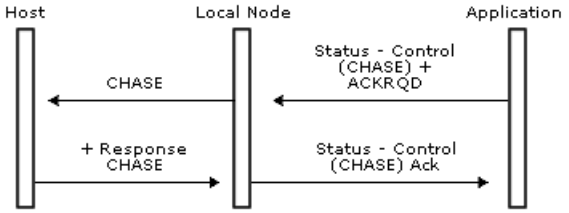
The use of ACKRQD on Status-Control messages effectively enforces definite-response and immediate request mode. This is appropriate for:

- **Status-Control** messages that correspond to the SNA requests CLEAR and STSN (because the expedited flow is RQD).
- Status-Control messages corresponding to all the DFC requests (which are RQD) except LUSTAT (which can be RQE).

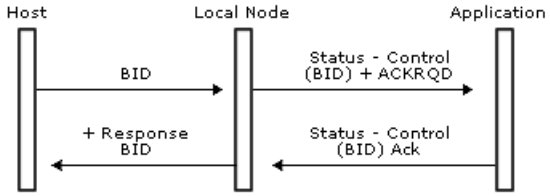
The application can set ACKRQD on **Status-Control** requests that correspond to SNA requests on the expedited flow, even where ACKRQD is not required. For example, when an application is signaling for direction (for example, a 3270 emulator with a terminal operator repeatedly pressing the ATTN key), it can generate multiple **Status-Control(SIGNAL) Request** messages, which would adversely affect the performance of other users. The application can set ACKRQD on the first **Status-Control(SIGNAL) Request** and ignore events that would cause further **Status-Control(SIGNAL) Request** messages until the **Status-Control(SIGNAL) Response** is received from the local node.

The message flows in the following six figures show outbound and inbound Status-Control sequences with and without ACKRQD and the corresponding SNA RUs:

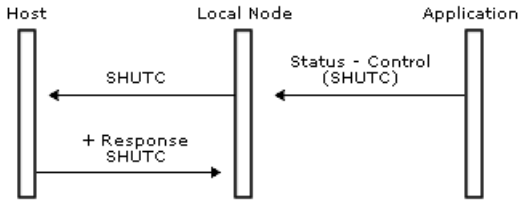
In the first illustration, the application sends Status-Control(CHASE).



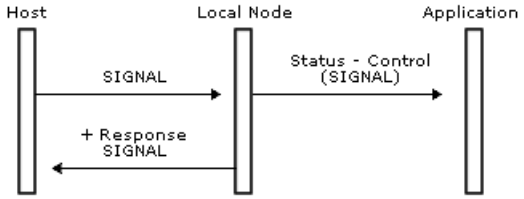
In the following illustration, the host sends BID request.



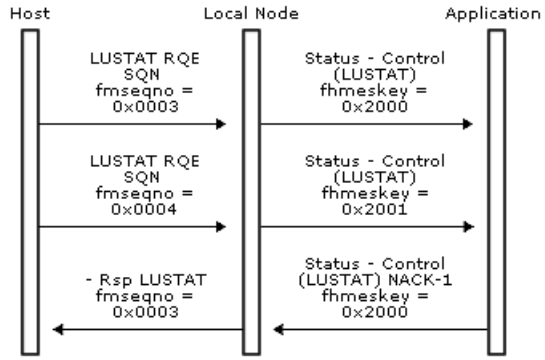
In the following illustration, the application sends **Status-Control(SHUTC)**.



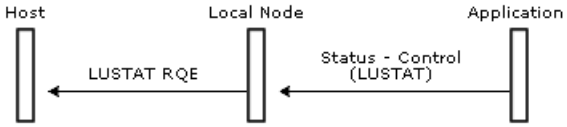
In the following illustration, the host sends SNA SIGNAL request.



In the following illustration, the host sends multiple RQE LUSTAT requests; the application rejects the first one.



In the following illustration, the application sends **Status-Control(LUSTAT)** NOACKRQD.



The following table summarizes the **Status-Control** requests supported by the local node and SNA session control (SC) and data flow control (DFC) requests. For each **Status-Control** request, the table gives:

- The SNA category of the corresponding SNA request (SC or DFC).
- The flow used by the corresponding SNA request (normal or expedited).
- The TS or FM profiles on which the corresponding SNA request is supported.
- The directions for which it is valid (NODE <–> APPL).
- Whether it requires ACKRQD; note that the application can set ACKRQD on a Status-Control request that does not require it.
- The hexadecimal code used in the control-type field of the Status-Control message (see FMI Message Formats).

Status-Control	SNA RQ flow	TS profile	FM profile	Direction node–appl	ACKRQD	Code
CLEAR	SC,Exp	2,3,4	–	–>	ACKRQD	CCLEAR (0x01)

SDT	SC,Exp	3,4	–	–>	–	CSDT (0x02)
RQR	SC,Exp	4	–	<–	–	CRQR (0x03)
STSN	SC,Exp	4	–	–>	ACKRQD	CSTSN (0x04)
CANCEL	DFC,Norm	–	3,4,7	<–>	ACKRQD	CCANCEL (0x10)
LUSTAT	DFC,Norm	–	3,4,7	<–>	–	CLUSTAT (0x11)
SIGNAL	DFC,Exp	–	3,4,7	<–>	–	CSIGNAL (0x12)
RSHUTD	DFC,Exp	–	3,4,7	<–	–	CRSHUTD (0x13)
BID	DFC,Norm	–	3,4	–>	ACKRQD	CBID (0x14)
CHASE	DFC,Norm	–	3,4	<–>	ACKRQD	CCHASE (0x15)
SHUTC	DFC,Exp	–	3,4	<–	–	CSHUTC (0x16)
SHUTD	DFC,Exp	–	3,4	–>	–	CSHUTD (0x17)
RTR	DFC,Norm	–	3,4	<–	ACKRQD	CRTR (0x18)
QC	DFC,Norm	–	4	<–>	ACKRQD	CQC (0x20)
QEC	DFC,Exp	–	4	<–>	–	CQEC (0x21)
RELQ	DFC,Exp	–	4	<–>	–	CRELQ (0x22)

The following requests are used only with LUA (see FMI Concepts).

Status-Control	**SNA RQ flow**	**TS profile**	**FM profile**	**Direction node–appl**	**ACKRQD**	**Code**
CRV	SC,Exp	3,4	–	–>	ACKRQD	CCRV (0x05)
BIS	DFC,Norm	–	18	<–>	ACKRQD	CBIS (0x19)
SBI	DFC,Exp	–	18	<–>	ACKRQD	CSBI (0x1A)

The use of particular **Status-Control** messages is described in following topics of this section, in the context of PLU session protocols such as chaining, brackets, recovery, and so on. See also Status-Control for the formats of **Status-Control** messages.

This section contains:

- Status-Control (ACKLUA) Message

Status-Control (ACKLUA) Message

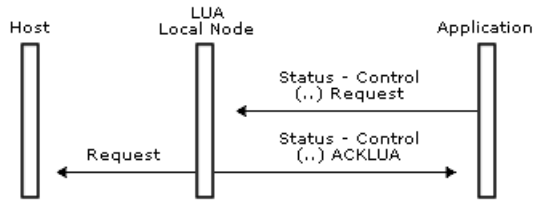
When an LUA application sends a **Status-Control** message inbound to the local node, the LUA verb used to send the message cannot complete until the local node acknowledges the message. Because of this, the local node will always respond to the LUA application after it has completed its send checking of the inbound message. If the inbound message passes the local node's send checks, and the corresponding SNA message will be sent to the host, the local node sends a Status-Control(...) ACKLUA message to the application to indicate this. Note that the ACKLUA message does not imply that the SNA message was successfully sent to the host, or that the host received it.

The format of the Status-Control(...) ACKLUA message is explained in Status-Control(...) ACKLUA. Note that the use of the message key field in Status-Control(...) ACKLUA is different from other Status-Control messages; it contains the sequence number from the TH of the Status-Control message sent by the LUA application, not the message key.

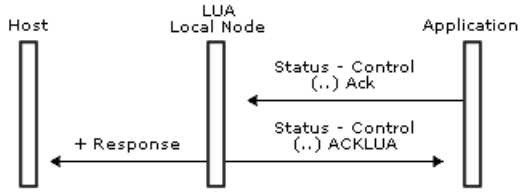
If the inbound message fails the local node's send checking, a Status-Control(...) Negative Acknowledge-2 message will be returned as for non-LUA applications. (This is then reported to the LUA application by a non-OK return code to the LUA verb that sent the message.)

The following three figures illustrate the ACKLUA acknowledgment protocol for different messages that the application can send.

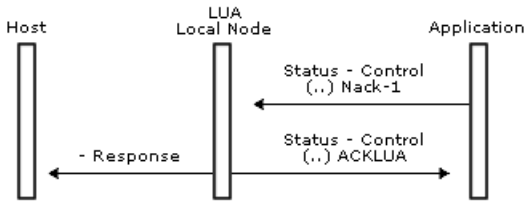
In the first illustration, the application sends a Status-Control(...) Request message that passes the local node's send checks.



In the following illustration, the application sends a **Status-Control(...) Acknowledge** message that passes the local node's send checks.



In the following illustration, the application sends a **Status-Control(...) Negative-Acknowledge-1** message that passes the local node's send checks.



Status-Error Message

Status-Error messages flow from the local node to the application to report request reject and RH usage error conditions for:

- Errors in outbound expedited data flow control (DFC) requests.
- Errors in outbound session control (SC) requests.
- Errors in inbound responses.

The **Status-Error** message contains four bytes of error code information that contain the appropriate SNA sense codes for the detected error. See Error and Sense Codes for a list of error codes.

Status-Resource Message

Status-Resource messages flow between an application and the local node to enable the local node to pace the application's PLU session. They provide the local node with an indication of the buffer resources available at the application to receive outbound messages. With this information, the local node can determine when to send a pacing response (see Pacing and Chunking).

Note that flow control is an option; inbound pacing is handled by the local node, transparently to the application, and outbound pacing can be handled similarly. This is appropriate when only a limited number of messages flows from end to end of the SNA session between definite responses, such as with a 3270 screen.

Status-Session Message

Status-Session messages always flow from the local node to the application and provide information about changes in the state of the session. There are separate **Status-Session** flows for each connection between the application and the local node.

The local node uses only one Status-Session message on the PLU connection; this is the Status-Session(BETB) message, used to report when the PLU session returns to the between-bracket state after the application or the PLU initiated a bracket (see Brackets).

The local node reports the activation and deactivation states of the SSCP session and PU-SSCP session using Status-Session messages. For example, it reports the receipt of an ACTLU request from the host SSCP using a Status-Session (LU-Active) message on the SSCP connection (see The SSCP Connection).

By providing Status-Session messages rather than requiring the application to interpret the relevant information in the SNA request, the local node shields the application from decisions affecting conditional state transitions and from the necessity for a detailed understanding of SNA protocols.

A Status-Session message contains a status code, and for some status codes, an additional status code that qualifies the meaning of the primary status code. For example, the Link-Error status code, which occurs on the SSCP connection, is qualified by a status code that reports the link outage code supplied by the data link control layer of the local node. Applications such as 3270 device emulators use the qualifying status codes to display communications check codes (–+z_nnn) on the display's status line.

The Status-Session codes are summarized in Status-Session Codes.

Status-RTM Message

The Status-RTM message is used by the local node to inform the application of the RTM (Response Time Monitor) parameters being used by the host. It flows from the local node to the application on the SSCP connection and is sent only for 3270 display LUs (or LUs in a pool of display LUs).

The Status-RTM message is sent at the following times:

- After the OK response to the Open(SSCP) Request message, to inform the application of the initial RTM parameters.
- When the RTM counters are reset, either due to a request from the host or when the local node sends unsolicited RTM data to the host.
- When any of the RTM parameters are changed by the host.

See RTM Parameters for more information on the use of the **Status-RTM** message and Response Time Monitor Data for more information on how the application supplies RTM data to the local node.

FMI Message Summary

This section lists each of the messages used at the FMI and gives a reference to a topic in the section where each message is described. The formats of the messages are given in FMI Message Formats.

For each message the direction of flow is indicated; IN means from the application to the local node, and OUT means from the local node to the application. The connection on which the message flows is also given.

Message	Direction	Connection	Reference
Open(SSCP) Request	IN	SSCP	Opening the SSCP Connection
Open(SSCP) OK Response	OUT	SSCP	Opening the SSCP Connection
Open(SSCP) Error Response	OUT	SSCP	Opening the SSCP Connection
Open(PLU) Request	OUT	PLU	Opening the PLU Connection
Open(PLU) OK Response	IN	PLU	Opening the PLU Connection
Open(PLU) Error Response	IN	PLU	Opening the PLU Connection
Open(PLU) OK Confirm	OUT	PLU	Opening the PLU Connection
Open(PLU) Error Confirm	OUT	PLU	Opening the PLU Connection
Close(SSCP) Request	IN	SSCP	Closing the SSCP Connection
Close(SSCP) OK Response	OUT	SSCP	Closing the SSCP Connection
Close(PLU) Request	IN/OUT	PLU	Closing the PLU Connection
Close(PLU) OK Response	IN/OUT	PLU	Closing the PLU Connection
Data-FMI	IN/OUT	SSCP/PLU	Data Flow
Status-Acknowledge(Ack)	IN/OUT	SSCP/PLU	Data Flow, Confirmation and Rejection of Data
Status-Acknowledge(Nack-1)	IN/OUT	SSCP/PLU	Data Flow, Confirmation and Rejection of Data
Status-Acknowledge(Nack-2)	OUT	SSCP/PLU	Inbound Data
Status-Control(CLEAR) Request	OUT	PLU	Recovery
Status-Control(CLEAR) Ack	IN	PLU	Recovery
Status-Control(CLEAR) Nack-1	IN	PLU	Recovery
Status-Control(SDT) Request	OUT	PLU	Status-Control Message
Status-Control(RQR) Request	IN	PLU	Recovery
Status-Control(RQR) Ack	OUT	PLU	Recovery
Status-Control(RQR) Nack-1	OUT	PLU	Recovery
Status-Control(RQR) Nack-2	OUT	PLU	Recovery
Status-Control(STSN) Request	OUT	PLU	Recovery
Status-Control(STSN) Ack	IN	PLU	Recovery
Status-Control(STSN) Nack-1	IN	PLU	Recovery
Status-Control(CANCEL) Request	IN/OUT	PLU	Outbound Chaining, Inbound Chaining
Status-Control(CANCEL) Ack	IN/OUT	PLU	Outbound Chaining, Inbound Chaining
Status-Control(CANCEL) Nack-1	IN/OUT	PLU	Outbound Chaining, Inbound Chaining
Status-Control(CANCEL) Nack-2	OUT	PLU	Inbound Chaining
Status-Control(LUSTAT) Request	IN/OUT	PLU	LUSTATs
Status-Control(LUSTAT) Ack	IN/OUT	PLU	LUSTATs
Status-Control(LUSTAT) Nack-1	IN/OUT	PLU	LUSTATs
Status-Control(LUSTAT) Nack-2	OUT	PLU	LUSTATs
Status-Control(SIGNAL) Request	IN/OUT	PLU	Direction
Status-Control(SIGNAL) Ack	OUT	PLU	Direction
Status-Control(SIGNAL) Nack-1	OUT	PLU	Direction
Status-Control(SIGNAL) Nack-2	OUT	PLU	Direction
Status-Control(RSHUTD) Request	IN	PLU	Application-Initiated Termination
Status-Control(RSHUTD) Ack	OUT	PLU	Application-Initiated Termination
Status-Control(RSHUTD) Nack-1	OUT	PLU	Application-Initiated Termination
Status-Control(RSHUTD) Nack-2	OUT	PLU	Application-Initiated Termination
Status-Control(BID) Request	OUT	PLU	Brackets
Status-Control(BID) Ack	IN	PLU	Brackets
Status-Control(BID) Nack-1	IN	PLU	Brackets

Status-Control(CHASE) Request	IN/OUT	PLU	Confirmation and Rejection of Data
Status-Control(CHASE) Ack	IN/OUT	PLU	Confirmation and Rejection of Data
Status-Control(CHASE) Nack-1	IN/OUT	PLU	Confirmation and Rejection of Data
Status-Control(CHASE) Nack-2	OUT	PLU	Confirmation and Rejection of Data
Status-Control(SHUTC) Request	IN	PLU	Shutdown and Quiesce
Status-Control(SHUTC) Ack	OUT	PLU	Shutdown and Quiesce
Status-Control(SHUTC) Nack-1	OUT	PLU	Shutdown and Quiesce
Status-Control(SHUTC) Nack-2	OUT	PLU	Shutdown and Quiesce
Status-Control(SHUTD) Request	OUT	PLU	Shutdown and Quiesce
Status-Control(RTR) Request	IN	PLU	Brackets
Status-Control(RTR) Ack	OUT	PLU	Brackets
Status-Control(RTR) Nack-1	OUT	PLU	Brackets
Status-Control(RTR) Nack-2	OUT	PLU	Brackets
Status-Control(QC) Request	IN/OUT	PLU	Shutdown and Quiesce
Status-Control(QC) Ack	IN/OUT	PLU	Shutdown and Quiesce
Status-Control(QC) Nack-1	IN/OUT	PLU	Shutdown and Quiesce
Status-Control(QC) Nack-2	OUT	PLU	Shutdown and Quiesce
Status-Control(QEC) Request	IN/OUT	PLU	Shutdown and Quiesce
Status-Control(QEC) Ack	OUT	PLU	Shutdown and Quiesce
Status-Control(QEC) Nack-1	OUT	PLU	Shutdown and Quiesce
Status-Control(QEC) Nack-2	OUT	PLU	Shutdown and Quiesce
Status-Control(RELQ) Request	IN/OUT	PLU	Shutdown and Quiesce
Status-Control(RELQ) Ack	OUT	PLU	Shutdown and Quiesce
Status-Control(RELQ) Nack-1	OUT	PLU	Shutdown and Quiesce
Status-Control(RELQ) Nack-2	OUT	PLU	Shutdown and Quiesce
Status-Error	OUT	SSCP/PLU	Status-Error Message
Status-Resource	IN	PLU	Pacing and Chunking
Status-Session	OUT	SSCP/PLU	Status-Session Message, Status-Session Codes
Status-RTM	OUT	SSCP	RTM Parameters

The following messages are used for LUA only:

Message	Direction	Connection	Reference
Status-Acknowledge(ACKLUA)	OUT	SSCP/PLU	Inbound Data from LUA Applications
Status-Control(...) ACKLUA	OUT	PLU	Inbound Data from LUA Applications
Status-Control(CRV) Request	OUT	PLU	Status-Control Message
Status-Control(CRV) Ack	IN	PLU	Status-Control Message
Status-Control(CRV) Nack-1	IN	PLU	Status-Control Message
Status-Control(BIS) Request	IN/OUT	PLU	Status-Control Message
Status-Control(BIS) Ack	IN/OUT	PLU	Status-Control Message
Status-Control(BIS) Nack-1	IN/OUT	PLU	Status-Control Message
Status-Control(SBI) Request	IN/OUT	PLU	Status-Control Message
Status-Control(SBI) Ack	IN/OUT	PLU	Status-Control Message
Status-Control(SBI) Nack-1	IN/OUT	PLU	Status-Control Message

FMI Status, Error, and Sense Codes

This section lists the status codes, error codes, and sense codes used on Open messages, Status messages, and Data messages with the system detected error indicator (SDI) set.

This section contains:

Status-Session Codes

For the status codes used on Status-Session messages, the following table lists:

- The value for the status code.
- The valid qualifying codes (if any) and their values.
- On which sessions the combinations of primary and qualifying status codes can occur.

See Status-Session Message for an overall description of the role of the **Status-Session** message. The individual codes are discussed in The SSCP Connection and The PLU Connection.

Status code	Value	Qualifying code	Value	Usage
STNOSESS (no session)	0x01	STPUINAC STPUACT STPUREAC STLUINAC	0x10 0x03 0x04 0x11	SSCP SSCP SSCP SSCP
STLINERR (link error)	0x02	DLC ERROR	(See Note 1)	SSCP
STLUACT (LU active)	0x05	-	-	SSCP
STLUREAC (LU reactivated)	0x06	-	-	SSCP
STBETB (between brackets)	0x07	-	-	PLU

Note The qualifying status code supplied on a Status-Session link error is the error code supplied by the data link control layer of the local node.

Status-Session link error code 20 is generated by the node rather than by the link service. It indicates that the link service is not yet available, but is being activated. Ignore this error code during session activation. Otherwise, all **Status-Session** link errors (including 20) will cause the emulator to send a **Close(SSCP)**. When the emulator receives the Close(SSCP) Response, it will start again and send a new **Open(SSCP)**.

Note The session status identifier displayed by the Microsoft® SNA Server OS/2 and Microsoft MS-DOS®-based 3270 emulators is obtained by adding 484 (decimal) to the qualifying status code shown above.

Error and Sense Codes

This section describes the error and sense codes that are reported to the application in the following messages:

- Open(SSCP) Response
- Open(PLU) Confirm
- Status-Acknowledge(Nack-2)
- Status-Control(...) Negative-Acknowledge-2
- Status-Error
- Appl-Data messages with SDI set.

Where the reported codes are SNA sense codes, a more complete description is given in Chapter 8 of the IBM document *Systems Network Architecture: Reference Summary* (GA27-3136). These SNA sense codes are also documented in the separate SNA Formats online Help file provided with Microsoft SNA Server.

In addition, the local node delivers negative responses from the host as Status-Acknowledge(Nack-1) and Status-Control(...) Negative-Acknowledge-1, which can have any SNA sense codes.

Application designers should note that error codes listed here that are specific to Host Integration Server and SNA Server always have an initial byte of value 0x00, and therefore can be easily distinguished from SNA sense codes, which have nonzero initial bytes.

The error codes are listed in topics for each type of message with an indication of the reason for the error.

This section contains:

Error Codes for Open Messages

The possible error codes for **Open(SSCP) Error Response** and **Open(PLU) Error Confirm** are shown in the following topics.

This section contains:

Error Codes for Open(SSCP) Error Response

The following table gives the values for error code 1 and error code 2 that can be returned on the **Open(SSCP) Error Response**.

Error code 1	Description	Error code 2	Description
0	No servers found.	CSRENOSR (0)	No servers found.
0x0053	LU not verified.	CSRECBSH (3)	LU not verified.
0x0055	SSCP connection already open.	CSRECBSH (3)	Control block / resource shortage.
0x0057	No LU in group free.		
0x0812	No free session control block available.		
0x1001	A connection activation failed recently.		
0x1002	The connection is inactive.		
0x1008	The link service is active remotely. This error return code is not supported by this level of Host Integration Server or SNA Server.		
0x1009	The SNA server is active, but the connection on which the requested LU is defined is not active.		
0x100B	The connection is in the process of activating as the result of another Open(SSCP) or operator activation or recovery from an outage.		
0x1010	The connection is active, but an ACTPU has not yet been received.		
0x1011	The connection is active, but an ACTLU has not yet been received.		
0x0063	Unrecognized Open request.	CSRESERV (1)	Service not present.
0x0A0E	LU / LU group not found in configuration.		

Note The error code 1 values 0x1001 to 0x1011 are returned when the Open(SSCP) Request specifies a nonforced Open. They do not indicate errors, but indicate that the LU-SSCP session is not active. The application can retry the **Open(SSCP) Request** specifying a forced Open, in which case the local node will attempt to activate the connection if possible.

Error Codes for Open(PLU) Error Confirm

The following table gives the values for error code 1 that can be returned on the Open(PLU) Error Confirm message. Error code 2 is zero, except when error code 1 is 0x0821; in this case it contains the byte offset in the BIND where the BIND failed to match the BIND check table.

Error code	Description
0x0051	Fewer than two buffer elements were present on **Open(PLU) Response**.
0x0052	SSCP connection no longer active.
0x0821	BIND checking failed: error code 2 gives byte offset in BIND at which the error occurred.

Error Codes for Nack-2 Messages

The possible error codes delivered to the FMI application on Status-Acknowledge(Nack-2) and Status-Control(...) Negative Acknowledge-2 messages are tabulated below. A Nack-2 is delivered to the application in response to data that is sent in error (or a Status-Control(...) Request that is in error). The data has not been sent to the host. The table indicates whether the error is critical or not (applying to the PLU connection only); if the error is critical, the critical failure indicator will be set in the message, and the application will receive a Close(PLU) Request as the next message.

All Nack-2 messages have the second word of information as 0x0000.

Error / Sense code	Critical YES/NO	Description
0x0040	YES	No buffer element on DATAFMI message.
0x0042	YES	DATAFMI message sent when no credit.
0x0043	YES	Invalid status-control for TS profile.
0x0044	YES	Invalid status-control from application.
0x004A	YES	HDX contention and -QR,-BB,EB, or BKTFSM in pending-term-session.
0x0809	YES	Mode inconsistency.
0x1002	YES	RU length error.
0x1003	YES	Function not supported, invalid FM profile.
0x2002	NO	Chaining error.
0x2003	NO	Bracket error.
0x2004	NO	Direction error
0x2005	YES	Data traffic reset.
0x2006	YES	Data traffic quiesced.
0x200D	YES	Response owed before sending request (half-duplex).
0x4003	YES	BB not allowed.
0x4004	YES	EB not allowed.
0x4006	YES	Exception response not allowed.
0x4007	YES	Definite response not allowed.
0x4009	YES	CD not allowed.
0x400A	YES	No-response not allowed.
0x400B	YES	Chaining not supported.
0x400C	YES	Brackets not supported.
0x400D	YES	CD not supported.
0x400F	YES	Incorrect use of FI.
0x4014	YES	Incorrect use of DR1, DR2, ER.
0x8005	NO	SSCP data sent when LU inactive.

Error Codes for Status-Error Messages

The possible error codes delivered to the FMI application on Status-Error messages are tabulated below. A **Status-Error** message is delivered to the application in one of several cases:

- The local node detects an error in a response sent from the application (as a **Status-Acknowledge** or **Status-Control Ack/Nack-1** message).
- The local node detects an error in some data from the host that will not be delivered to the application as an SDI message (such as an expedited flow request).
- The application sends an invalid Status message.

For inbound responses, the **Status-Error** codes have first byte 0x00. When the application is in error, the table indicates whether the error is critical or not (applying to the PLU connection only); if the error is critical, the application will receive a Close(PLU) Request as the next message.

The sense codes beginning with 0x40 will only be delivered if the corresponding receive check has been enabled in the CICB on the Open(SSCP) Request from the application.

Where the sense code is marked with the * symbol, the second word of sense information carries the request code of the expedited flow request that was in error (for example 0x00C9 for SIGNAL).

Error / Sense code	Critical YES/NO	Description
0x0008	NO	Negative response already sent to this chain.
0x0040	YES	Invalid **Status** message from application.
0x0046	YES	Session failure due to correlation table shortage.
0x0050	YES	Invalid sequence number on Status-Ack.
0x0053	YES	Application may not send status control (STSN) negative acknowledge if it supports transaction numbers.
0x0056	YES	Status-Ack sent when previous RQD chains are outstanding (see Outbound Data).
0x0801	NO	Message received when pacing count is zero.
0x0805	NO	BIND from another PLU when already bound.
0x0809 *	NO	Mode inconsistency (QEC or SHUTD).
0x0815	NO	BIND from same PLU when already bound.
0x0821	NO	Incorrect ACTLU type (SSCP connection).
0x1003 *	NO	Wrong profile / network control request / invalid session control message.
0x2005	NO	Data traffic reset.
0x2007	NO	Data traffic not reset (STSN after SDT).
0x4009 *	NO	CD not allowed.
0x400B *	NO	Chaining not supported.
0x400C *	NO	Brackets not supported.
0x400F *	NO	Incorrect use of FI.
0x4011 *	NO	Incorrect use of RU category.
0x4014 *	NO	Incorrect use of DR1, DR2, ER.

Sense Codes for SDI Messages

When the local node detects an error in a normal flow request from the host, the message is converted into a DATAFMI message with the system detected error indicator (SDI) set to inform the application and to allow data to be processed serially. The application must convert the message to a Status-Acknowledge(Ack) to allow the local node to send the required negative response to the host. The possible error codes delivered to the FMI application on such SDI messages are tabulated below.

The sense codes beginning with 0x40 will only be delivered if the corresponding receive check has been enabled in the CICB on the Open(SSCP) Request from the application. If a receive check has been disabled, the message can still be converted to an SDI message; for example, a message with BB, -BC would fail as 2002 or 2003 if 4003 were disabled.

When the application uses a Status-Control(LUSTAT) Request to reject outbound data (see LUSTATs), the sense codes supplied by the application will be present on the SDI message generated by the local node.

Sense code	Description
0x0809	Mode inconsistency.
0x080B	Bracket race error.
0x081B	Contention race condition.
0x1003	Incorrect FM profile for request.
0x2001	Sequence number error.
0x2002	Chaining error.
0x2003	Bracket error.
0x2004	Direction error.
0x2006	Data traffic quiesced.
0x4003	BB not allowed.
0x4004	EB not allowed.
0x4006	Exception response not allowed.
0x4007	Definite response not allowed.
0x4009	CD not allowed.
0x400B	Chaining not supported.
0x400C	Brackets not supported.
0x400D	CD not supported.
0x400F	Incorrect use of FI.
0x4011	Incorrect use of RU category.
0x4014	Incorrect use of DR1, DR2, ER.

Configuration Information

To obtain information about the Microsoft® Host Integration Server or Microsoft® SNA Server 3270 configuration, the application uses the following calls:

sepdcrec	Returns a data structure that contains the 3270 user record for this user and the diagnostics record from the running configuration file.
sepdgetinfo	Returns general information on the version of Host Integration Server or SNA Server currently running, such as the release level, the network operating system, and the directory of the running configuration file.

If the return code from **sepdcrec** indicates that no 3270 user record was found for this user, the emulation program should terminate and not allow the user to use 3270 emulation. The Host Integration Server or SNA Server error message COM0438 is provided to log this error; see Diagnostics for more information on error logging.

3270 User Record Format

The format of the 3270 user record is shown below. The first structure definition is an LU/session information record, which includes details of a 3270 LU; the second is the 3270 user record, which includes a number of LU/session information records.

Note that the user record is not a fixed length, because the number of LU/session information records in the remap list is variable. The structures used below are provided simply as a template to allow you to map to the correct offset in the record.

```
typedef struct tecwrksd {
    UCHAR   cwshost[9];
    USHORT  cwsestyp;
    USHORT  cwsmodov;
    USHORT  cwspad;
} TECWRKSD;
```

Members

cwshost[9]
LU/pool name accessed.
cwsestyp
Session type (M2, M3, M4, M5, printer).
cwsmodov
Whether or not the user has override permission.
cwspad
Two bytes of padding.

```
typedef struct tecwrkus {
    USHORT   cwlen;
    USHORT   cwtype;
    UCHAR    cwname[21];
    UCHAR    cwremark[26];
    UCHAR    cwstylef[9];
    USHORT   cwvewrtm;
    USHORT   cwalert;
    USHORT   cwchghan;
    USHORT   cwmaxses;
    USHORT   cwnumrec;
    TECWRKSD cwsesdat[10];
    USHORT   cwmodisf;
    UCHAR    cwstatus;
    UCHAR    cwpad;
    USHORT   cwnumrmp;
    TECWRKSD cwremap[1];
} TECWRKUS;
```

Members

cwlen
Length of record.
cwtype
Type of record.
cwname[21]
User name.
cwremark[26]
Comment field.
cwstylef[9]
Initial style file name.
cwvewrtm
Whether or not user can view RTM information.
cwalert
Whether or not user has ALERT permission.
cwchghan
Whether or not user can change LU/pool name accessed.

cwmaxses
Maximum number of active sessions (1–10).
cwnumrec
Number of sessions for user.
cwsesdat[10]
Session information records.
cwmodisf
Permission to modify initial style.
cwstatus
Status byte: user or group.
cwpad
One byte of padding.
cwnumrmp
Number of LUs/pools in remap list.
cwremap[1]
LU/Pool remap list.

For Windows 2000, Windows NT, Windows 98, and Windows 95

Microsoft® Host Integration Server or Microsoft® SNA Server permit configuration of more than 10 sessions per user when used with clients running Windows 2000, Windows NT, Windows 98, and Windows 95. If this is done, the first 10 sessions are placed in the **cwsesdat** array with **cwnumrec** set to 10 and the remainder are placed in the location of the remap list. The **cwnumrmp** member indicates the number of **TECWRKSD** structures in the remap list. Note that this permits **cwmaxses** to be greater than **cwnumrec**.

The following paragraphs explain the meaning of each field in the structures, and indicate how the application should use it. The sections on configuring 3270 users and LUs in the Installation and Configuration section of the *Microsoft Host Integration Server Guide* should also be used as a cross-reference.(see the *Microsoft SNA Server Administration Guide* for SNA Server).

Members

cwshost
The name (up to eight characters) of the LU or LU pool that this session is configured to use. The application specifies this name on the Open(SSCP) Request.
cwsestyp
The LU type (display or printer) of the LU used by this session and (if it is a display LU or a pool of display LUs) the screen model. The possible values are:

CERTMOD2 (0) Model 2 display (24 by 80)
CERTMOD3 (1) Model 3 display (32 by 80)
CERTMOD4 (2) Model 4 display (43 by 80)
CERTMOD5 (3) Model 5 display (27 by 132)
CERTPRNT (4) Host printer

The application should use this value to distinguish between display and printer sessions and to set the appropriate screen model for display sessions.

cwsmodov
TRUE if the user has permission to override the screen model for display sessions — that is, to change the session to use a different screen model from the one configured. If this value is FALSE, the user should not be permitted to change the screen model. This field is not used for printer sessions and should not be checked.
cwlen
The length of the 3270 user record (this is variable because it contains a variable number of LU/session records in the remap list). The application should use this value to locate the start of the next 3270 user record when searching for the correct record.
cwtype
Identifies this as a 3270 user record.
cwname
The LAN Manager user name, or other identifying name, of the 3270 user (up to 20 characters). The application uses this to search for the correct 3270 user record.
cwremark
An optional comment field (up to 25 characters), used in the configuration program to give more information about the user

(for example, the user's full name).

cwstylef

The name (up to eight characters) of the default style file used by this user (a file containing the user's 3270 customization settings, used by the Host Integration Server or SNA Server 3270 emulation programs). This field can be used to identify the equivalent file for your 3270 emulator, if appropriate.

If this field is blank, no style file is used and the 3270 emulator should revert to its default settings (unless overridden by a style file specified by the user).

cwvewrtm

TRUE if this user is permitted to view a display of Response Time Monitor (RTM) statistics for his or her 3270 sessions. If this field is FALSE, the application should not display RTM statistics and should not display a last transaction time indicator (LTTI) on the status line of display sessions. See Diagnostics Record Format for more information on the use of RTM.

cwalert

TRUE if the user is permitted to send NetView user alerts. If this field is FALSE, the user should not be permitted to send alerts. See Diagnostics Record Format for more information on the use of alerts.

cwchghan

TRUE if the user is permitted to remap a 3270 session to use a different LU (in which case it can be changed to use any LU in the remap list—see **cwremap**). If this field is FALSE, the application should not allow the user to remap sessions.

cwmaxses

The maximum number of active sessions permitted to this user. If the number of sessions configured (see **cwnumrec**) is greater than this, the user must not be allowed to activate more sessions at a time than this field specifies.

cwnumrec

The total number of sessions configured for this user. The user record always contains 10 LU/session records (see **cwsesdat**), but only this number of the records will be used—the remainder will be filled with zeros.

cwsesdat

Ten LU/session records. Some of these records can be filled with zeros, indicating that they are unused (**cwnumrec** gives the number of sessions that are used). The application should list, and allow the user to use, only the sessions that have valid session records here.

cwmodisf

TRUE if the user is permitted to modify the initial 3270 customization. If this field is FALSE, the application should use the customization defined by **cwstylef** (if specified); the user should not be allowed to make changes to this style, or to override it by loading a different style file.

cwstatus

Indicates whether the user name in this record is to a LAN Manager user name or group name. The least significant bit of this byte is CERTGRUP (1) for a group, and zero for a user. Other bits are not used.

cwpad

Pad byte—not used by the application.

cwnumrmp

The number of LU/session records in the remap list (see **cwremap**).

cwremap

The list of LU/session records, which indicates the LUs to which the user can remap sessions (if any). If the user is not permitted to remap sessions (see **cwchghan**), this list is not used and should not be checked by the application.

Diagnostics Record Format

The format of the Diagnostics record is shown below. The first structure definition is an Alert information record, which includes details of a 3270 NetView user alert; the second is the Diagnostics record, which includes a number of Alert information records.

```
typedef struct tedalert {
    UCHAR    dalrtnam[53];
    UCHAR    daparam1[33];
    UCHAR    daparam2[33];
    UCHAR    daparam3[33];
} TEDALERT;
```

Members

dalrtnam[53]
Description of the alert number.
daparam1[33]
Description of parameter 1.
daparam2[33]
Description of parameter 2.
daparam3[33]
Description of parameter 3.

```
typedef struct tediagns {
    USHORT   dilen;
    USHORT   ditype;
    UCHAR    dinetmgt[9];
    USHORT   disrtmco;
    USHORT   disrtmub;
    USHORT   diwruldr;
    USHORT   dirtmth1;
    USHORT   dirtmth2;
    USHORT   dirtmth3;
    USHORT   dirtmth4;
    TEDALERT dialerts[20];
    UCHAR    diaudit[128];
    UCHAR    dierror[128];
    USHORT   diaudlev;
    UCHAR    dipad[16];
} TEDIAGNS;
```

Members

dilen
Length of record.
ditype
Type of record.
dinetmgt[9]
Network management connection name.
disrtmco
Send RTM data at counter overflow.
disrtmub
Send RTM data at UNBIND.
diwruldr
RTM timers run until:
dirtmth1
RTM threshold #1.
dirtmth2
RTM threshold #2.
dirtmth3
RTM threshold #3.
dirtmth4

RTM threshold #4.

dialerts[20]
Alert description records.

diaudit[128]
Audit log file name.

dierror[128]
Error log file name.

diaudlev
Default audit level.

dipad[16]
Sixteen bytes of padding.

The following paragraphs explain the meaning of each field in the structures that is relevant to the application and indicate how the application should use it. Fields that are not included in the list below are used by other Microsoft® Host Integration Server or Microsoft® SNA Server components and need not concern the application; in particular, the network management connection name and the times at which RTM data is sent to the host are handled by the local node on behalf of the application.

Note that the application should determine whether the user is permitted to send NetView user alerts and/or view RTM data (see 3270 User Record Format); it should not display the appropriate information, as described below, if the user does not have permission to use it. The host can also override whether the application is permitted to send and/or to display RTM data (see RTM Parameters for more information).

For more information on how the application uses the RTM parameters, see RTM Parameters, Response Time Monitor Data, and Status-RTM.

Members

dalrtnam
The description (up to 52 characters) of the alert corresponding to a particular alert number. The application should display this information to help the user determine which alert to send.

daparam1
The descriptions (each up to 32 characters) of up to three parameters.

daparam2
Required for the alert; depending on the specific alert.

daparam3
One or more of these descriptions can be blank, indicating that the parameter is not used. For each of these descriptions that is not blank, the application should display this string to prompt the user for the appropriate parameter.

diwruldr
The definition by which response times are to be measured. The application should measure the response time from the time the user presses ENTER or an AID key to send data to the host, until one of the following events as defined by this field:

CERTWRIT (0) The first host data reaches the 3270 display.

CERTUNLK (1) The host unlocks the user's keyboard.

CERTDIRE (2) The host gives the application direction so that the user can send further data.

Note that the host can override these definitions; see RTM Parameters for more information.

dirtmth1, **dirtmth2**, **dirtmth3**, **dirtmth4**
The thresholds that define the bands into which response times are to be classified. Note that the host can override these definitions; see RTM Parameters for more information.

dialerts
Up to 20 alert records that define the alerts Host Integration Server or SNA Server users can send to a host. There are always 20 records, but some of these can be blank, indicating that they are not used; the application should display the descriptions of any nonblank alerts together with the alert number (from 1 to 20) defined by the position of the alert record in this array.

Creating NetView User Alerts

You can create NetView user alerts for users to send. Users identify the alerts by number; each number corresponds to a specific collection of information or requests that the user wants to send through NetView to a host operator.

Microsoft® Host Integration Server and Microsoft® SNA Server leave blank fields for the user alert information in the structure that is returned from sepdcrec. To create specific user alerts, create appropriate data structures and call the TRANSFER_MS_DATA common service verb to send the user alert to NetView.

Diagnostics

This section describes the diagnostics mechanisms available to Microsoft® SNA Server applications.

This section contains:

- Error and Audit Log Messages
- Internal Tracing
- HLLAPI Parameter Tracing
- FMI Tracing

Error and Audit Log Messages

This section discusses ways that an application can write to the Microsoft® Host Integration Server or Microsoft® SNA Server log files, and describes macros for logging and tracing information.

Options for Logging

There are two ways in which an application can write messages to the Microsoft® Host Integration Server or SNA Server log files:

- Use the SNA server DMOD (dynamic access module) logging macros presented in the following topic.
- Use the LOG_MESSAGE common service verb.

Both of these options use log message files formatted with the OS/2 utility MKMSGF. A log message file contains a set of messages that are referenced by a message number. If a component wishes to log an error, it specifies the appropriate message number to extract the text from the message file, rather than including the message text within the component.

Host Integration Server and SNA Server have one message file that is used for all its error and audit logging. The messages in this file are available for both of the preceding options. See the Administrator's Reference section of the *Microsoft Host Integration Server Guide* for a list of all the SNA server error and audit log messages and their meanings (see the *Microsoft SNA Server Reference* for SNA Server).

If extra messages are required, there are two options:

- The messages COM0393 and COM0394 are provided as generic log messages for use by emulators. Each takes two parameters, both of which are text strings: the first is an identifier for the specific emulator that logged the message, and the second can contain any data or parameters to be logged. The difference between these messages is the level at which they are logged; COM0393 is a level 10 information message, while COM0394 is a warning message that should be used to report error conditions. See SNA Server DMOD Logging Macros for more information on message severity levels.
- A second message file can be created, which is then specified as one of the parameters on the **LOG_MESSAGE** verb. This cannot be used with the logging macros.

It is recommended that you use the Host Integration Server or SNA Server logging macros (with the Host Integration Server or SNA Server emulator log messages if a suitable message is available, or with the generic messages COM0393 and COM0394) rather than the **LOG_MESSAGE** verb. **LOG_MESSAGE** does not provide control over the severity level at which messages are logged (all messages are logged at level 12 in the error log file); it is primarily intended for user applications such as APPC TPs.

The following topics explain the use of the Host Integration Server or SNA Server logging macros and the **LOG_MESSAGE** common service verb.

SNA Server DMOD Logging Macros

The logging macros provided with Host Integration Server or SNA Server are relatively easy to use because each log call is a single line of code. In the simplest case, only a message number is required; the text of the logged message is taken from the message file. Parameters can also be supplied as required.

For the text and meaning of each of the log messages included in the SNA Server message file, see the See the Administrator's Reference section of the *Microsoft Host Integration Server Guide* (see the *Microsoft SNA Server Reference* for SNA Server). Examples of messages logged by a Host Integration Server or SNA Server 3270 emulation program are shown in Examples.

Message Severities

The following severities are used in Host Integration Server and SNA Server:

6	Detailed problem analysis data
8	General information messages
10	Significant system events
12	Warnings/recoverable errors
16	Fatal errors

All logs are placed in the Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, or Microsoft® Windows® 95 Application event log. The default level of event logs can be specified when configuring Host Integration Server or SNA Server, so that all or some of the event logs can be suppressed; in addition, the system administrator can change the event log level for a particular component using the Manage program or can filter out lower-level messages when viewing log files using the Browse program.

Note that level 16 errors are always taken to be fatal errors; an application that logs a level 16 error will be terminated automatically. In particular, the call to the logging routine will not return; the application should perform all required clean-up processing, including any lower-level logs, before logging the fatal error.

Logging Macros

The following macros can be used to log messages at levels 6, 8, 10, 12, and 16:

COM_LOG6
COM_LOG8
COM_LOG10
COM_LOG12
COM_LOG16

Syntax

COM_LOG*a* (*b* **)** "" **))**
for a message with no parameters.

COM_LOG*a* (*b* **)** " %*c* " , *e* **))**
for a message with one parameter.

COM_LOG*a* (*b* **)** " %*c* | %*d* | . . ." , *e* , *f* , . . .**))**
for a message with more than one parameter.

Parameters

a
Severity: 6, 8, 10, 12, or 16.
b
Message number.
c, *d*, . . .
Format of the first, second, and so on up to nine variable parameters.
e, *f*, . . .
First, second, and so on up to nine variable parameters.

Remarks

Up to nine parameters can be supplied, according to the number of "*%n*" placeholders in the text of the message being logged. The first parameter replaces %1, the second replaces %2, and so on.

The formats *c* and *d* must be valid formats for the C function **sprintf**, because the logging macro uses this function to generate the complete text string to be logged.

Note that the unmatched parentheses on the macro call are deliberate; the expansion of the macro supplies the remaining parentheses so that the resulting code is correct.

Examples

This syntax is illustrated in the following examples. The two messages below are taken from the Host Integration Server or SNA Server message file; the first is a warning message with no parameters, and the second is a level 16 error message that includes two parameters.

```
COM0392W: Warning – User Alert was not accepted by the host
COM0429E: Error trying to read from %1 (rc = %2)
```

The following call:

```
COM_LOG12(392)""));
```

gives the following message in the event log:

```
Warning – User Alert was not accepted by the host
```

The following call:

```
COM_LOG16(429)"%s|%d", cfgfilename, rc));
```

where *cfgfilename* = COM.CFG, and *rc* = 17, gives the following message in the event log:

```
Error trying to read from COM.CFG, rc = 17
```

The complete message log entry contains the following information about the error or event and the service that logged it:

Message Number
Date/Time
COM name the name of the service
COM type the type of the service
Severity
Message with optional parameters

The COM name is the name in the service table when the DMOD initializes; this name is the name with which the user logged on to the network operating system and is set by calling sepdcrec. The COM type is set on the call to sbpuinit. See The DL-BASE/DMOD Interface for more information on these functions.

LOG_MESSAGE Common Service Verb

This verb is fully documented in LOG_MESSAGE in the APPC Applications section of the *Microsoft Host Integration Server Developer's Guide*.

The main advantage of using this verb to log messages is that a user-defined message file can be specified, containing messages specific to your application. This means that you are not restricted to the messages defined in the Host Integration Server or SNA Server message file.

Note that all messages logged using this verb are logged at level 12 in the error log file; it is not possible to log messages to the audit log file or to log at different severity levels.

Internal Tracing

Microsoft® Host Integration Server and Microsoft® SNA Server provide internal tracing macros that can be built into an application. These can be used to check the operation of the application during development. Compile-time options determine whether or not this tracing is included in object code; it should be compiled out when producing end-user products. Internal tracing must be initialized before any **TRACEnn** macro calls are issued. This initialization is performed by initializing the DMOD (dynamic access module) through the sbpuinit call. Internal Tracing Macros and Controlling Internal Tracing describe the use of the tracing macros within application source code and the methods of controlling trace output using compile-time and run-time options.

Internal Tracing Macros

The following topics describe macros used for internal tracing:

- COM_ENTRY
- TRACEn

COM_ENTRY

The **COM_ENTRY** macro is used at the start of any procedure in which internal tracing is required. It provides an identifier that is used in the trace file to identify all trace calls made from this procedure.

The format of the **COM_ENTRY** call is as follows:

COM_ENTRY("*str*"**)**;

where *str* is a string of up to five characters that uniquely identifies this procedure.

TRACEn

The **TRACE*n*** macro is used to specify data to be traced; this data can include variable parameters. The format of the **TRACE*n*** call is one of the following, depending on whether parameters are included:

TRACE*n*()"*string in* ***sprintf*** *format*"**))**;
TRACE*n*()"*string in* ***sprintf*** *format containing parameters*", *parameters***))**;

Note that the unmatched parentheses are deliberate; they are resolved by the expansion of the macro.

The value *n* identifies the severity level of the trace. It can take the values 2, 4, 6, 8, 10, 12, or 16, where levels 6 to 16 correspond to the audit and error log levels (see Error and Audit Log Messages), and 2 and 4 are used for very low-level detail tracing. This allows run-time filtering of trace information so that only information above a specified level is logged; see the following topic for more information.

The following examples illustrate the use of the tracing macros:

```
COM_ENTRY("proc1");
TRACE4()"Start of error-handling routine"));
TRACE8()"Supplied parameters are %s, %d", parm1, parm2));
```

Controlling Internal Tracing

The compiler option **NOTRC** defines whether internal tracing is included in object code. Compiling with **/DNOTRC** does not include internal tracing (the **TRACE*n*** macros expand to a no-op); compiling without **/DNOTRC** includes internal tracing.

Under Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95, the registry entries used to control tracing are now inserted by the Setup program. The SNATRACE.EXE program can be used to enable or disable internal tracing dynamically at run time (assuming binaries have been compiled with internal tracing enabled).

For Windows 2000, Windows NT, Windows 98, and Windows 95

When running an executable program that was compiled with internal tracing, tracing is enabled by generating the following entries in the Windows 2000, Windows NT, Windows 98, or Windows 95 registry:

```
InternalTraceLevel=n
InternalTraceFile1=file1
InternalTraceFile2=file2
```

The entries should be stored under:

**HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Nap
\Parameters**

For OS/2

When running an executable program that was compiled with internal tracing, the COMINT environment variable is used to define the level of tracing required and the file or files to which it is traced. Set this variable as follows:

SET COMINT=*n*/*file1*[/*file2*]

For all operating systems

The value *n* is the severity level of the tracing required. All trace calls at this level or higher are included in the trace output; trace calls at lower levels are ignored. For example, setting level 10 includes all trace calls at levels 10, 12, and 16, but excludes tracing at levels 2, 4, 6, and 8. Use 0 to include all tracing at whatever level, or 20 to disable tracing entirely.

The parameters *file1* and *file2* are the names of files to which trace output is written. If two file names are specified, trace output is sent to the first file until it reaches 250K (or 500K for Windows 2000, Windows NT, Windows 98, and Windows 95) and then to the second file; when the second file also reaches 250K (or 500K for Windows 2000, Windows NT, Windows 98, and Windows 95), the first file is cleared and tracing continues to the first file. This process continues, changing to the other file every time the current file reaches 250K (or 500K for Windows NT and Windows 95), so that only the most recent 250-500K (or 500-1000K for Windows 2000, Windows NT, Windows 98, and Windows 95) of trace information is retained. If only one file name is specified, tracing continues to this file regardless of file size.

HLLAPI Parameter Tracing

The **COM_TRC_HLLAPI** macro can be used within an HLLAPI (High-Level Language API) library to trace the parameters supplied to an HLLAPI function and the returned values. It should be used at the following times:

- At the start of the HLLAPI library routine, to trace the four parameters supplied by the HLLAPI application before beginning to process them.
- At the end of the HLLAPI library routine, to trace the values to be returned in each of the four parameters before control is returned to the application.

The format of the **COM_TRC_HLLAPI** call is as follows:

COM_TRC_HLLAPI (*type***,** *func***,** *data***,** *len***,** *retc***);**

Parameters

type (**USHORT**)
Either REQUEST or RESPONSE; use REQUEST when tracing the parameters supplied by the user application at the start of the library routine and RESPONSE when tracing the parameters to be returned to the user application.

func, *len*, and *retc* (**USHORT**)
The values of the parameters *func_number*, *data_length,* and *return_code* that were supplied to HLLAPI by the user application, or the values to be returned to the application. Note that the user application supplies pointers to these values, but the actual values are required for the **COM_TRC_HLLAPI** call.

data (**UCHAR FAR**)
The *data_string* parameter supplied to HLLAPI by the user application, or the string to be returned to the application in this parameter.

Remarks

To turn on tracing for an HLLAPI application, the environment variable COMTRC is used; see the Administrator's Reference section of the *Microsoft Host Integration Server Guide* for more information on the use of COMTRC (see the *Microsoft SNA Server Administration Guide* for SNA Server). Tracing can also be turned on and off from within the HLLAPI user application using the trace control parameters of the Set Session Parameters function; see the HLLAPI Specification for more information.

The trace output lists the following:

- The name of the HLLAPI function being called (this is determined from the function number, the first of the four parameters to the HLLAPI call).
- Whether the trace refers to a request or a response (determined from the *type* parameter to the **COM_TRC_HLLAPI** call).
- The *data_length* and *return_code/ps_position* parameters to the HLLAPI call.
- If the *data_string* parameter is used for this HLLAPI function, the address of the data as supplied to HLLAPI, followed by a listing of the data in both hexadecimal and ASCII characters.

Note that Microsoft® SNA Server's HLLAPI tracing is designed for use with the SNA Server implementation of HLLAPI on OS/2, which is compatible with the implementation provided by OS/2 Extended Edition version 1.2. If your HLLAPI implementation differs from this (for example, by including function numbers that are not supported by the SNA Server implementation, by using a different name for a given function number, or in the usage of the *data_string* parameter for different verbs), the tracing output obtained may not be correct.

For Microsoft® Windows 2000, Microsoft® Windows NT®, Microsoft® Windows® 98, and Microsoft® Windows® 95, you can use SNATRACE.EXE to dynamically control HLLAPI and FMI tracing.

FMI Tracing

Microsoft® SNA Server provides the facility for tracing message flows at the Function Management Interface (FMI), both at the local node and at the application's DL-BASE. This allows you to track the messages being sent and received by the local node and/or the application.

For Windows 2000, Windows NT, Windows 98, and Windows 95

Message tracing at the application's DL-BASE is controlled by entries in the registry under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Nap\Parameters

Message tracing at the local node is controlled by placing similar entries under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Node\Parameters

The entries required in both cases are as follows:

```
MessageTraceFile1=file1
MessageTraceFile2=file2
DLCTraceState=on (or off)
```

SNATRACE.EXE can be used to dynamically control HLLAPI and FMI tracing.

For OS/2

Message tracing is controlled by the COMSNA environment variable. You set this variable before starting the local node using the COMM START command, or before starting the client application; the application must be started in the screen group in which COMSNA was set.

The syntax of COMSNA is described in the *Microsoft SNA Server Administration Guide*. To activate FMI tracing, you need to use one of the options **/A** (all messages are traced) or **/F** (only FMI messages are traced).

For the local node, you can leave tracing initially inactive by not specifying any of the final options on COMSNA, and then activate it when required using the Manage program; in this case you need to use the internal message tracing option in Manage, which is the equivalent of the **/A** option on COMSNA. For the application, you need to specify either **/A** or **/F** to activate tracing at start of day, because it cannot be activated from the Manage program.

You can also use the Manage program to start the local node with tracing, instead of setting COMSNA and then starting the local node. Again, you need to use the internal message tracing option, which is the equivalent of COMSNA with the **/A** option.

Compiling and Linking 3270 Client Applications

This section describes the 3270 client samples included with the Microsoft® Host Integration Server 2000 and the Microsoft SNA Server SDK. These samples are also supplied as part of the Microsoft Developer Network (MSDN®) Platform SDK. The samples are located in the \SDK\Samples\SNA folder on the Host Integration Server 2000 CD-ROM (these samples are located under the \SDK\SAMPLES folder on the earlier SNA Server CD-ROM).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\SNA subdirectory below where the MSDN Platform SDK has been installed.

This section lists and explains the header files and libraries needed to develop 3270 client applications for use with Host Integration Server 2000 and SNA Server client applications. The section also provides information on compiling and linking the 3270 client applications.

This section contains:

Building the 3270 Client Samples

When installed from the Host Integration Server CD-ROM, all the 3270 client samples are built in a similar way, as described in this section. First, set the following environment variables:

Variable	Description
ISVLIBS	The directory containing the Microsoft Host Integration Server 2000 LIB files for Microsoft Windows® 2000, Window NT®, Windows 98, and Windows 95.
ISVINCS	The directory containing the Host Integration Server 2000 header files.
SAMPLEROOT	The root directory where the sample code provided as part of the SDK has been installed on a local hard disk.

For example, after copying the contents of the SDK folder on the Host Integration Server 2000 CD-ROM to C:\SNASDK, use the following lines to set the variables (assumes Intel binaries are being produced for Windows 2000, Windows NT on I386, Windows 98, or Windows 95):

```
ISVLIBS=C:\SNASDK\Lib
ISVINCS=C:\ SNASDK\Include
SAMPLEROOT=C:\SNASDK\Samples\SNA
```

Next, run NMAKE on the .mak file in each subdirectory below this root directory containing the actual sample source code. For example, for APING and APINGD, change to the Samples\aping directory and type the following:

nmake -f makeping.mak

Note that Windows NT on DEC Alpha is not supported by the Host Integration Server SDK. If you wish to build these samples on Windows NT for DEC Alpha, the earlier SNA Server 4.0 SDK will be required for accessing the Windows NT import libraries for DEC Alpha under the \SDK\LIB\WINNT\ALPHA folder.

When these samples are installed as part of the Microsoft Developer Network (MSDN) Platform SDK, the build process is simpler and doesn't require that any environment variables be set. Open an MS-DOS Command Windows, navigate to the HIS\SNA folder, and run NMAKE to build all of the SNA samples. To compile a specific sample (RUI3270, for example), navigate to the appropriate subdirectory (SNA\RUI3270, for example) and run NMAKE.

Client Interface Files for 3270 Applications

The following files are required to build 3270 client applications for use with Microsoft Host Integration Server or Microsoft SNA Server:

File	Description
FMI.H	Main header file containing the definitions of buffer and message formats, function prototypes for the DL-BASE/DMOD interface calls, and constant definitions.
TRACE.H	Definitions of the logging and tracing macros (see Diagnostics for more information).
FMISTR32.LIB	Function Management Interface string library for use with Windows 2000, Windows NT, Windows 98, or Windows 95.
SNACLI.LIB	Main interface library for developing 3270 client applications on Windows 2000, Windows NT, Windows 98, or Windows 95.

The following additional files supplied with the SNA Server SDK are required to build 3270 client applications for Microsoft Windows 3.x, MS-DOS, or OS.2:

File	Description
COMCLI.LIB	Main interface library for developing 3270 client applications on OS/2.
DOSACS.LIB	Main interface library for developing 3270 client applications on Microsoft MS-DOS®.
FMISTR.LIB	Function Management Interface string library for use with Microsoft Windows 3.x.
WINCLI.LIB	Main interface library for developing 3270 client applications on Microsoft Windows 3.x.

3270 Include Files

To compile the application, the header files FMI.H and TRACE.H are required. In addition, one of the standard operating system header files may be required. To include the required files, the following lines should be used in your application:

For Windows 2000, Windows NT, Windows 98, and Windows 95

```
#include <fmi.h>
#include <trace.h>
```

For Windows version 3.*x*, and MS-DOS

```
#include <fmi.h>
#include <trace.h>
```

For OS/2

```
#include <os2.h>
#include <fmi.h>
#include <trace.h>
```

Compiler Options for 3270 Applications

When compiling the 3270 client application, the following compiler options are required:

Option	Explanation
/c	Compile only, without linking. Linking is normally done as a separate phase to include the required Host Integration Server or SNA Server libraries.
/D NOTRC	The **NOTRC** macro specifies that internal tracing should not be compiled into the application. See Diagnostics for more information on the use of internal tracing. The /D NOTRC option should be used for building a final system (internal tracing should not be included because it will degrade performance and require more memory and resources). For a development system, you may want to compile with internal tracing; if so, remove the /D NOTRC option.
/D WIN32_SUPPORT /D MSWIN_SUPPORT, /D OS2_SUPPORT, /D DOS_SUPPORT	These macros are used in the header files FMI.H and TRACE.H supplied with SNA Server to support variants of the client interface for the different operating systems supported. One of these options must be defined, depending on the operating system for which the application is intended.

For Windows 2000, Windows NT, Windows 98, and Windows 95

/Gzs	c:	Use stdcall calling conventions on i386/i486 and Pentium class processors.
	S:	Remove stack check calls.
Note:		
/Gcs	c:	On MIPS, Alpha, and PowerPC processors, use Pascal calling conventions.

For Windows 3.*x*, OS/2, and MS-DOS

/G2cs	2:	Use 286 instructions.
	c:	Use Pascal calling conventions.
	s:	Remove stack check calls.
/Zp	Structures must be packed.	
/J	Default character type is unsigned. Not valid under Windows 2000, Windows NT, Windows 98, and Windows 95.	

For all operating systems

The following compiler flags are required, but any of the valid options for each flag may be used, as appropriate to your application:

/A	Compiler model (for Windows 3.x, MS-DOS, and OS/2 applications only)
/O	Optimization
/W	Warning level

Linking 3270 Client Applications

The following describes how to link 3270 client applications using different platforms.

For Windows 2000, Windows NT, Windows 98, and Windows 95

The SNACLI.LIB library must be linked with the application.

The DMOD is implemented as a DLL. SNACLI.LIB contains import definitions for the APIs in the DLL, and some global variables required for the logging and tracing macros.

For OS/2

The COMCLI.LIB library must be linked with the application.

The DMOD is implemented as a DLL. COMCLI.LIB contains import definitions for the APIs in the DLL, and some global variables required for the logging and tracing macros.

For Windows 3.*x*

The WINCLI.LIB and WLOGTR.LIB libraries must be linked with the application.

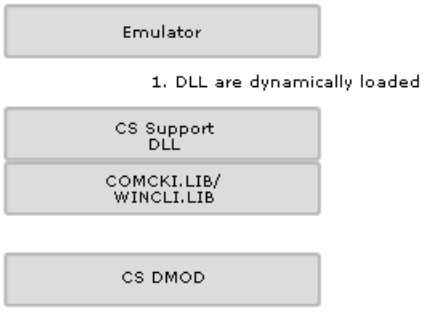
The DMOD is implemented as a DLL. WINCLI.LIB contains import definitions for the APIs in the DLL, and some global variables required for the logging and tracing macros. WLOGTR.LIB contains import definitions for log and trace routines.

For MS-DOS

The DOSACS.LIB library must be linked with the application. It contains DL_BASE, DMOD, and diagnostics code.

For all operating systems

It is possible to create a DLL that is dynamically loaded when the user starts a session for an LU. In this case, to make the log and trace macros available, the application structure needs to be as shown below:



Support for 3270 Single Sign-On

This section describes the support for single sign-on for 3270 display sessions that is available in Microsoft® Host Integration Server 2000 and in Microsoft® SNA Server version 3.0 with Service Pack 1 or higher.

Over 3270 LUs, a single sign-on feature is supported to automate the overall logon process. When configured for this feature, Host Integration Server or SNA Server automatically replaces special keywords in the data stream with the actual host user name and password at appropriate points in the session.

Prerequisites for 3270 Single Sign-On

In preparation for using 3270 single sign-on, the system administrator must define a host security domain containing host connections. This host security domain must be initially created or modified to enable the single sign-on feature. The system administrator must enable a user's Microsoft® Windows 2000 or Microsoft® Windows NT® account in the host security domain and either the administrator or the user must establish a mapped host account for the Windows 2000 or Windows NT domain user name.

The user must be logged on to a Windows 2000 or Windows NT domain with a user name and password. Note that this single sign-on feature is only supported over 3270 LUs.

Registry Settings Used for 3270 Single Sign-On

The 3270 single sign-on feature depends on Host Integration Server or SNA Server scanning 3270 LUs used in the logon process for special keywords that are defined in the registry on the computer running Host Integration Server or SNA Server. The values for these special keywords can be defined by the system administrator on the computer running Host Integration Server or SNA Server.

The registry settings used by the 3270 single sign-on process are located under the **HKEY_LOCAL_MACHINE\CurrentControlSet\Services** registry node. Installed under the **SNASERVR\PARAMETERS** subkey are the following entries:

3270SSOPadByte
This entry should be set to an ASCIIZ string to use as the character for padding replacement text in the user name or password if these strings are shorter than the length of the special tag strings defined below. The default value for this pad character is the ASCII space character.

3270SSOPostReplaceCount
This entry should be set to a DWORD that represents the number of message chains of RUs to scan after replacement of text for user name or password. The default value for this number is 10.

3270SSOPrefix
This entry should be set to an ASCIIZ string to use as the special prefix tag string in combination with the user name and password tags. The default value of this string is MS$.

3270SSOPwdTag
This entry should be set to an ASCIIZ string to use as the special tag string in combination with the **3270SSOPrefix** tag in defining the special host password string that will be replaced. The default value of this string is SAMEP, so the default host password string that is scanned for and replaced is MS$SAMEP. Note that length of the password string that is scanned for (MS$SAMEP, for example) determines the maximum length of the password string that can sent to the host using single sign-on. This limit occurs because the password substitution cannot change the length of the data message
Note that the value of this string must be different from the value of the **3270SSOUserTag** entry for single sign-on to function properly.

3270SSOReplaceCount
A DWORD value that affects the timeout value for password substitution. User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the timer expires. By default the timer will be set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node.

If the **3270SSOReplaceCount** registry entry is defined and the **3270SSOReplaceTimer** registry entry is not defined, the node counts this number of RUs (on PLU-SLU session only) before timeout occurs. If both the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries are defined, the value for **3270SSOReplaceCount** will be used to determine when a timeout will occur. By default, this key is not defined and the node defaults to a timeout of 30 seconds.

3270SSOReplaceTimer
A DWORD value that affects the timeout value for password substitution. User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the timer expires. By default the timer will be set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node.

If the **3270SSOReplaceTimer** registry entry is defined and **3270SSOReplaceCount** is not defined, the node uses this value in seconds before timeout occurs. If both the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries are defined, the value for **3270SSOReplaceCount** will be used to determine when a timeout will occur. By default, this key is not defined and the node defaults to a timeout of 30 seconds.

3270SSOUserTag
This entry should be set to an ASCIIZ string to use as the special tag string in combination with the **3270SSOPrefix** tag in defining the special user name string that will be replaced. The default value of this string is SAMEU, so the default user name string that is scanned for and replaced is MS$SAMEU.
Note that length of the user name string that is scanned for (MS$SAMEU, for example) determines the maximum length of the username string that can sent to the host using single sign-on. This limit occurs because the username substitution cannot change the length of the data message
Note that the value of this string must be different from the value of the **3270SSOPwdTag** entry for single sign-on to function properly.

3270 User Name and Password Replacement

The SNA node on the host monitors the inbound session for a replacement sequence consisting of the **3270SSOPrefix** string immediately followed by one of the strings **3270SSOUserTag** or **3270SSOPwdTag**. Thus, the default user name string that would be scanned for and replaced is MS$SAMEU. When this string is found in the inbound session data, the node looks up the corresponding information (host user name in the current host security domain) and overwrites MS$SAMEU with this information. The same process occurs for the password string that would be scanned for and replaced, which defaults to MS$SAMEP.

Note that this operation cannot change the length of the data message. If the actual user name or password that is retrieved from the current host security domain is shorter than the replacement sequence, it is padded out with the first character of the **3270SSOPadByte** string used as a padding character. If the actual host user name or password string is longer than the string that is scanned for, these strings are truncated to the length of the scanned string so that the data message length is not affected.

Note that since the username and password can be sent in any order, the registry string values for the **3270SSOUserTag** and **3270SSOPwdTag** entries must be different for single sign-on to function properly.

The SNA node monitors the SSCP-LU session for these special tag strings at all times and replaces all occurrences of these strings with corresponding looked-up data. On the LU-LU session, the node starts monitoring at start of session (BIND). The node stops monitoring when it has received **3270SSOPostReplaceCount** chains of RUs without seeing a substitution tag. The node will not restart monitoring until it receives an UNBIND–BIND sequence for that session.

Note that the node considers the sequence:

```
BIND, data, UNBIND(BIND FORTHCOMING), BIND …
```

as a continuation of the same LU-LU session and does not restart monitoring on receipt of the second BIND. This sequence is often used by host session managers handing off a session to an application subsystem, and is considered a single terminal session.

User IDs and passwords will be substituted in each chain on the LU-SSCP and PLU-SLU sessions until the node has received **3270SSOPostReplaceCount** chains of RUs without seeing a substitution tag or a timer expires. By default the timer is set to 30 seconds, but this behavior can reconfigured in the registry using the **3270SSOReplaceCount** and **3270SSOReplaceTimer** registry entries. The timer is started when the OPEN SSCP is received by the node. After the timer expires, the node will stop scanning messages for the 3270 replacements strings for the user ID and password. If the replacement strings arrive after the timer expires, the replacement strings will be sent to the host unmodified causing the signon to fail. The application will not receive any notification that the timer has expired. The only indication of a problem will likely be that the signon to the host session has failed.

Note that all strings are specified in the registry in ASCII, but the node translates them to EBCDIC through AE character mapping before scanning for a match.

3270 Emulation Reference

This section contains the reference material for the 3270 Emulator.

This section contains:

DL-BASE/DMOD Entry Points

This section gives definitions for the entry points to the DL-BASE and DMOD.

CMDGoTSR

The **CMDGoTSR** function initiates a background thread for the emulator and then executes an MS-DOS TSR interrupt.

MS-DOS

```
SHORT APIENTRY CMDGoTSR(
    ULONG entryPoint,
    UCHAR FAR *stack,
    UCHAR FAR *topOfRam
   );
```

Parameters

entryPoint
Pointer to the function where the background thread will start execution.

stack
Pointer to the stack of the background thread.

topOfRam
Top of RAM; all memory above this address will be released by the NAP for LAN Manager or the NAP for NetWare (LMBASE and NWBASE, respectively).

Remarks

An MS-DOS-based emulator should complete its initialization, then execute this call to go resident. A thread of execution will be created at the entry point specified.

This call will never return control to the calling program.

CMDSemClear

The **CMDSemClear** function clears a RAM semaphore.

MS-DOS

```
USHORT FAR CMDSemClear(
    ULONG FAR *ramSem
   );
```

Parameters

ramSem
Address of the semaphore.

CMDSemRequest

The **CMDSemRequest** function requests a RAM semaphore.

MS-DOS

```
USHORT FAR CMDSemRequest(
    ULONG FAR *ramSem,
    ULONG timeOut
   );
```

Parameters

ramSem
Address of the semaphore.
timeOut
Length of time in milliseconds to wait before returning.

Return Values

0
OK.
ERROR_SEM_TIMEOUT
Time-out expired before semaphore operation completed.
ERROR_SEM_OWNED
This thread or another thread owns the semaphore, and the calling thread specified zero time-out.

CMDSemSet

The **CMDSemSet** function sets a RAM semaphore.

MS-DOS

```
USHORT FAR CMDSemSet(
    ULONG FAR *ramSem
   );
```

Parameters

ramSem
Address of the semaphore.

CMDSemWait

The **CMDSemWait** function waits until a RAM semaphore is cleared.

MS-DOS

```
USHORT FAR CMDSemWait(
    ULONG FAR *ramSem,
    ULONG timeOut
   );
```

Parameters

ramSem
Address of the semaphore.
timeOut
Length of time in milliseconds to wait before returning.

Return Values

0
OK.
ERROR_SEM_TIMEOUT
Time-out expired before semaphore operation completed.
ERROR_SEM_OWNED
This thread or another thread owns the semaphore, and the calling thread specified zero time-out.

CMDStartFG

The **CMDStartFG** function requests that scheduling of the foreground thread be resumed.

MS-DOS

```
USHORT FAR CMDStartFG();
```

Return Values

0
The foreground thread was successfully stopped.
nonzero
The foreground thread could not be stopped.

Remarks

The emulator should issue this call after restoring the screen contents when returning to background operation.

CMDStopFG

The **CMDStopFG** function requests that the foreground thread be suspended.

MS-DOS

```
USHORT FAR CMDStopFG(
    USHORT timeOut
   );
```

Parameters

timeOut
Maximum time to wait for the foreground thread to return from MS-DOS before stopping it.

Return Values

0
The foreground thread was successfully stopped.
nonzero
The foreground thread was stopped within MS-DOS.

Remarks

The emulator should issue this call when it wishes to enter the foreground. If the return value is nonzero, the foreground thread has been stopped within MS-DOS and it is not safe for the emulator to come to the foreground. Under these circumstances, the emulator should restart the foreground thread by calling CMDStartFG.

If the call was successful, the emulator should save the contents of the screen before writing to it. When returning to background operation, the emulator must restore the screen and call **CMDStartFG** to allow the previous foreground application to continue.

RegisterSwitchProc

The **RegisterSwitchProc** function registers an application procedure that will be called whenever the 3270 emulator is about to be switched in or out of memory by the MS-DOS version 5, MS-DOS version 6, or Windows 3.*x* task-switching code.

MS-DOS

```
USHORT FAR RegisterSwitchProc(
    ULONG switchProc
   );
```

Parameters

switchProc
A far pointer to the function where task activity will be notified. The function is defined as follows:

```
VOID FAR PASCAL SwitchProc(
    USHORT inOut
   );
```

Parameters

inOut
Zero if emulator is about to be switched out of memory, 1 if emulator is about to be switched back into memory.

routproc

The **routproc** function is a sample routing procedure. It must be supplied as part of the application. It is called by the DMOD with a message that may or may not be for this application; the DMOD calls routing procedures in turn until one accepts the message.

Win32

```
DWORD routproc(
    BUFHDR *msgptr,
    USHORT locl,
    USHORT retstat
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
USHORT FAR _loadds routproc(
    BUFHDR FAR *msgptr,
    USHORT locl,
    USHORT retstat
   );
```

Parameters

msgptr
Pointer to the message passed by the DMOD to the routing procedure.

locl
Locality from which message was received (if *retstat* indicates message returned), or locality to which path was lost (if *retstat* indicates path error).

retstat
Reason for call:

CEDINMSG (1)—message returned.

CEDINLLN (2)—path error (see Remarks below).

Return Values

TRUE
The routing procedure has accepted the message.

FALSE
The message is not for this routing procedure.

Remarks

The routing procedure should first call sbpurcvx, which handles any Open response messages, as follows:

sbpurcvx(&*msgptr*, *locl*, *retstat***)**

A return code of TRUE from **sbpurcvx** indicates that **sbpurcvx** has accepted the message; an Open error response has been received for this application, and resource location is continuing. The routing procedure should not process the message any further and should return TRUE to prevent the DMOD from calling further routing procedures.

A return code of FALSE from **sbpurcvx** indicates that the routing procedure should:

- If the message is for this application, take responsibility for the message and return TRUE to prevent the DMOD from calling further routing procedures.
- If the message is not for this application, return FALSE so that the DMOD tries further routing procedures.

If a path error is returned, *msgptr* will not point to a valid message, and no more FMI messages will be returned for the locality value indicated. The application is responsible for ending all sessions using this locality. The routing procedure must return FALSE; this ensures that the lost locality is reported to all other routing procedures.

If the message is for this application, the routing procedure can either process the message immediately or put the message on an application queue and then post the application using a semaphore. See Receiving Messages for more information.

sbpibegt

The application calls the **sbpibegt** function to get a buffer element to append to an existing buffer.

Win32

```
VOID sbpibegt(
    PTRBFELT *eltptr
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sbpibegt(
    PTRBFELT FAR *eltptr
   );
```

Parameters

eltptr
Pointer to a pointer to an element. On return this points to a pointer to the element obtained, or to NULL if an element was not obtained (an internal error).

Remarks

This function should only be used to get extra elements for an existing buffer. The sepdbubl function should be used to get a new buffer.

The new element should be added to the chain of elements from the existing buffer header and the count of the number of elements updated.

This function is typically used when a received buffer is being reused to transmit a message that is longer than the incoming message.

sbpiberl

The application calls the **sbpiberl** function to release a buffer element from an existing buffer.

WIn32

```
VOID sbpiberl(
    PTRBFELT *eltptr
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sbpiberl(
    PTRBFELT FAR *eltptr
   );
```

Parameters

eltptr
Pointer to a pointer to the element to be released.

Remarks

This function should only be used to release surplus elements from a buffer. The sepdburl function should be called to release the entire buffer.

The released element should first be removed from the element chain and the count of the number of elements updated.

This function is typically used when a received buffer is being reused to transmit a message that is shorter than the incoming message.

sbpuinit

The **sbpuinit** function initializes the DL-BASE.

Win32

```
USHORT sbpuinit(
    HANDLE *sema4ptr,
    USHORT proctype,
    USHORT servtype,
    UCHAR *uname
   );
```

MS-DOS

```
USHORT FAR sbpuinit(
    HSEM FAR *sema4ptr,
    USHORT proctype,
    USHORT servtype
   );
```

OS/2

```
USHORT FAR sbpuinit(
    HSEM FAR *sema4ptr,
    USHORT proctype,
    USHORT servtype,
    UCHAR *auname
   );
```

Windows Version 3.x

```
USHORT FAR sbpuinit(
    USHORT proctype,
    USHORT servtype,
    UCHAR FAR *uname
   );
```

Parameters

sema4ptr
Semaphore, created by DMOD, cleared by DMOD when a message is available. For MS-DOS, the application should supply the address of a 4-byte (long) integer. This address is for internal use by Host Integration Server or SNA Server—the application should not subsequently attempt to reference the address.

proctype
Type of process: CLIENT–2.

servtype
Type of service/client: CES3270–2.

uname
Pointer to a character buffer of length at least 21 characters; the LAN Manager user name, or other identifying name appropriate to the network operating system, is returned to the application in this buffer. The application does not need to use this parameter, but can use it for display or logging.

Return Values

NO_ERROR
Initialization successful.

Any other return value indicates that the initialization failed. This is usually an operating system return code. The following values are also used:

DMLTABF (555)
L table is full.

DMMNWGI (562)
Failed to get network operating system information.
DMDSTFL (563)
Service table is full.
DMMPIPF (567)
Failed to make a named pipe.
DMCOMNM (582)
No name registered for this application.
DMCOMDUP (596)
A service is already running with the same name.
DMNOTLOG (598)
User is not logged on to network operating system.
DMCFGOPN (616)
Failed to open configuration file.
DMCFGREAD (618)
Failed to read from configuration file.
DMNONAP (625)
The NAP is not started.
DMMAXAPP (953)
Windows only:
Maximum number of concurrent applications exceeded.

Remarks

The **sbpuinit** entry point should always be called before any other DL-BASE/DMOD entry points except SNAGetVersion. For new emulators, sepdcrec should be called after **sbpuinit**. (Because of the order of calls used in older emulators, a call to **sepdcrec** before **sbpuinit** is still supported, but this order is not recommended.)

sbpurcvx

The **sbpurcvx** function processes Open responses from a routing procedure. An application can define a routing procedure that is called by the DMOD when a message is received. This routing procedure should first call **sbpurcvx** to handle any Open response messages received. This ensures that Open responses intended for the Resource Locator are handled correctly.

Win32

```
USHORT sbpurcvx(
    BUFHDR * *msgptr,
    INTEGER locl,
    INTEGER retstat
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
USHORT FAR sbpurcvx(
    BUFHDR FAR * FAR *msgptr,
    INTEGER locl,
    INTEGER retstat
   );
```

Parameters

msgptr
Pointer to the message returned by the DMOD to the routing procedure.

locl
Locality from which message was received (if *retstat* indicates message returned), or locality to which path was lost (if *retstat* indicates path error).

retstat
Reason for call:

CEDINMSG (1—message returned.

CEDINLLN (2)—path error.

Return Values

TRUE
The Resource Locator has accepted the message; the application should not process it any further.

FALSE
The message should be processed by the application.

Remarks

This function is called by a routing procedure that is called by the DMOD; it is not called directly by the application.

The parameters for **sbpurcvx** should be taken from the parameters for routproc. Note, however, that the first parameter to **sbpurcvx** is a pointer to a pointer to a buffer header (that is, a pointer to the corresponding parameter for the routing procedure, not the parameter itself).

sbpusend

The **sbpusend** function sends a message from an application to a partner on an LPI connection.

Win32

```
VOID sbpusend(
    PTRBFHDR msgptr
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sbpusend(
    PTRBFHDR msgptr
   );
```

Parameters

msgptr
Pointer to the message to be sent.

Remarks

The message buffer is released after transmission by the DMOD. It cannot be accessed by the application again.

For an Open request message, the **destl** parameter can be zero. In this case, the Resource Locator will attempt to find a suitable destination for the Open message.

sbputerm

The **sbputerm** function must be called when the application terminates. It frees the DL-BASE/DMOD resources used by the application.

For Win32, do not call **sbputerm** from an entry point in a detached DLL process because it may cause a deadlock inside the SNADMOD.DLL.

Win32

```
VOID sbputerm(void);
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sbputerm(void);
```

sepdbubl

The application calls the **sepdbubl** function to get a buffer with a requested number of elements.

Win32

```
PTRBFHDR sepdbubl(
    USHORT noelts
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
PTRBFHDR FAR sepdbubl(
    USHORT noelts
   );
```

Parameters

noelts
Number of elements required.

Return Values

A pointer to the buffer obtained; NULL if a buffer could not be obtained.

Remarks

Each element has a size of 268—the constant SNANBEDA in the header file FMI.H.

The returned buffer consists of a header and the required number of elements. The header points to the first element, which points to the next element, and so on to make an element chain.

It is possible to add an element to an existing buffer by calling sbpibegt to get the extra element. The new element should be added to the element chain of the buffer, and the "number of elements" count should be updated.

The application must release any buffers that are not transmitted.

sepdburl

The application calls the **sepdburl** function to release a buffer.

Win32

```
VOID sepdburl(
    PTRBFHDR msgptr
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sepdburl(
    PTRBFHDR msgptr
   );
```

Parameters

msgptr
Pointer to the buffer to be released.

Remarks

It is important that buffers are released after use. This is done automatically when a message is transmitted. For messages received, it is the responsibility of the application to either release or reuse the buffer.

This function releases both the buffer header and any associated buffer elements. It is possible to release single elements from a buffer by using the function sbpiberl.

sepdchnk

The **sepdchnk** function gets the FMI chunk size. The application calls this function to obtain the chunk size that should be used on the FMI. See Pacing and Chunking for more information on FMI chunking. This feature is supported in Comm Server version 1.2 and later and in Host Integration Server and SNA Server by 3270 client applications developed for use on Windows 2000, Windows NT, WIndows 98, and Windows 95.

Win32

```
VOID sepdchnk(
    USHORT *pipesizeptr,
    USHORT *chunksizeptr
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
VOID FAR sepdchnk(
    USHORT FAR *pipesizeptr,
    USHORT FAR *chunksizeptr
   );
```

Parameters

pipesizeptr
Size in bytes of the pipe between the application and the local node.

chunksizeptr
DMOD chunk size in bytes.

Remarks

The application does not need to use the pipe size returned by this call. (It is included on this call because the local node uses the same call to obtain both the pipe size and the chunk size.)

sepdcrec

The **sepdcrec** function gets configuration information. The application calls this function to obtain the 3270 configuration information for the name with which the user logged on to the network operating system. The call also registers this user name in the service table.

Win32

```
USHORT sepdcrec(
    UCHAR *pBuffer,
    USHORT length,
    USHORT *numbytes
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
USHORT FAR sepdcrec(
    UCHAR *pBuffer,
    USHORT length,
    USHORT *numbytes
   );
```

Parameters

pBuffer
Pointer to a buffer supplied by the application, in which configuration information is returned.
length
Size of the supplied buffer.
numbytes
Used by Host Integration Server or SNA Server to return the number of bytes of information returned in the buffer.

Return Values

NO_ERROR (0)
OK.
NOCSSRVR (1)
No configuration file server available.
NODGNREC (2)
No diagnostics record found in configuration file.
NOUSRREC (3)
No user record found in configuration file for this user.
BUF2SMAL (4)
Supplied buffer was too small.
NONOS (5)
Network operating system is not started.
NOTLOGON (6)
User is not logged on to the network operating system.
READERR (7)
Failed to read from configuration file.
NONAP (8)
The NAP is not started.
MAXAPP (9)
Windows only:
Maximum number of concurrent applications exceeded.
ERROR_SERVER (14)
Error on the server end of the RPC.
ERROR_LOCAL_FAILURE (15)
Error on the local end of the RPC.

Remarks

The sbpuinit function should always be called before any other DL-BASE/DMOD entry points except SNAGetVersion. For new emulators, **sepdcrec** should be called after **sbpuinit**. (Because of the order of calls used in older emulators, a call to **sepdcrec**

before **sbpuinit** is still supported, but this order is not recommended.)

On successful return, the buffer contains pointers to the appropriate 3270 user record and the diagnostics record, followed by the records themselves. It is formatted as follows:

Win32

```
     TECWRKUS *pUserRecord,
     TEDIAGNS *pDiagRecord
    );
```

(UserRecord—variable length)

(DiagRecord)

MS-DOS, Windows Version 3.x, and OS/2

```
     TECWRKUS FAR *pUserRecord,
     TEDIAGNS FAR *pDiagRecord
    );
```

(UserRecord—variable length)

(DiagRecord)

The two records should be accessed using the supplied pointers.

See Configuration Information for details of the format of these records and of how the application uses the configuration file information.

If there is no 3270 user record for this user in the configuration file, or if no diagnostics record is found in the configuration file (an internal error), the application should terminate and not allow the user to use 3270 emulation. The Host Integration Server or SNA Server error log messages COM0438 and COM0437 can be used to report these failures.

If the supplied buffer is too small for the returned information, the contents of the buffer are undefined and should not be examined, but the *numbytes* parameter will contain the total number of bytes of information available (that is, the size of the two pointers plus the two configuration records). The application should retry with a buffer of at least this size.

sepdgetinfo

The **sepdgetinfo** function returns a structure containing the version number of Host Integration Server or SNA Server, the path of the current configuration file, and the network operating system over which SNA server is running.

Win32

```
USHORT sepdgetinfo(
    struct cs_info *pCSInfo
   );
```

MS-DOS, Windows Version 3.x, and OS/2

```
USHORT FAR sepdgetinfo(
    struct cs_info FAR *pCSInfo
   );
```

Parameters

pCSInfo
Pointer to a buffer supplied by the application, containing a **cs_info** data structure in which system information is returned. The application must set the **length** member in this data structure (see Remarks below); the other members should be set to nulls or blanks.

The cs_info structure

The returned **cs_info** structure and its members are as follows:

```
struct cs_info {
    unsigned short length;
    unsigned char  major_ver;
    unsigned char  minor_ver;
    unsigned char  config_share[80];
    unsigned short nos;
 } cs_info;
```

Members

length
Length of the data structure supplied by the application.

major_ver
Major version number:

1 for CS 1.1
2 for CS 2.0

minor_ver
Minor version number (decimal):

10 for CS 1.1 (indicates 1.10)
00 for CS 2.0 (indicates 2.00)

config_share[80]
Path of the running configuration file：\\server\share\ (null terminated).

nos
Network operating system in use

1: LAN Manager / LAN Server
2: NetWare

Return Values

NO_ERROR (0)
OK.

NOCSSRVR (1)
No configuration file server available.
BADLNGTH (2)
Supplied buffer was too small.

Remarks

The application must set the **length** member to the length of the **cs_info** structure (86 bytes in the current version). Any other value will be rejected. This parameter is used to ensure compatibility with future versions; an application supplying this length will always obtain the information shown here, but in future versions it may be possible to specify larger values and obtain further information.

On successful return, the data structure **cs_info** contains the version number of Host Integration Server or SNA Server, the path of the current configuration file, and the network operating system over which SNA server is running.

Do not use the configuration file path returned by **sepdgetinfo** because NetWare clients will not be able to access this path.

If there is no configuration file server available, only the version number fields are valid; the other fields should not be checked.

sepdrout

The **sepdrout** function for Win32, MS-DOS, and OS/2 allows an application to perform its own routing of received messages by setting up a procedure that is called by the DMOD when a message is received.

Win32

```
‹DWORD sepdrout(
DWORD ( *proc_addr,)
(BUFHDR *, USHORT, USHORT
);
 ›
DWORD sepdrout(
    DWORD *proc_addr,
    (BUFHDR *, USHORT, USHORT
   );
```

MS-DOS and OS/2

```
‹USHORT FAR sepdrout(
USHORT (FAR *FAR proc_addr)();
 ›
USHORT FAR sepdrout(

    USHORT (FAR *FAR proc_addr)();
   );
```

Parameters

proc_addr
The routing procedure.

Return Values

NO_ERROR (0)
Successful.
Anything else
Unsuccessful.

Remarks

This facility is only available to clients, as defined in the call to sbpuinit.

An application can have up to four routing procedures. Note that the APPC and CSV libraries each use a routing procedure. When the DMOD receives a message, each routing procedure is called, until one accepts the message.

See routproc for an example of a routing procedure.

sepwrout

The **sepwrout** function is the Windows version 3.*x* version of sepdrout; it has the same parameters and is used in exactly the same way.

SNAGetVersion

The **SNAGetVersion** function returns the major version number in the low byte and minor version number in the high byte.

MS-DOS

```
USHORT FAR SNAGetVersion(void);
```

FMI Message Formats

This section describes the message formats for the Function Management Interface (FMI). The message formats are presented in a language-independent notation. Details of the message format notation and key assumptions about the contents of the message formats are as follows:

- "Reserved" indicates that the field is set to zero (for a numeric field) or all nulls (for names) by the sender of the message.
- "Undefined" indicates that the value of the field is indeterminate. The field is not set by the sender and should not be examined by the receiver of the message.
- Fields that occupy two bytes — such as **opresid** in the Open(PLU) Request — are represented with the most arithmetically significant byte in the lowest byte address, irrespective of the normal orientation used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. However, the following fields are exceptions:

  The **srci** and **desti** fields in buffer headers are stored in the local format of the application that assigns them, since only the assigning application needs to interpret these values.

  The **startd** and **endd** fields in elements are always stored in low-byte, high-byte orientation (the normal orientation of an Intel processor).

- Messages are composed of buffers consisting of a buffer header and zero or more buffer elements; see Messages for more information on buffer formats.
- Applications must assign unique index (I) values for every active LPI connection within the node. In particular, the Open(SSCP) Request must be different from the source index it sends in response to the Open(PLU). Additionally, zero should not be used as an I value. An I value of zero means that the sender of the message is inviting the recipient of the message to assign an I value.
- The **startd** field in each element gives the offset of the first byte of data in the element after the **trpad** field.

  For non-LUA applications, **startd** will either be 1 (data starts in the byte after the **trpad** field), 10 (nine bytes of padding are included between the **trpad** field and the start of the data), or 13 (12 bytes of padding are included between the **trpad** field and the start of the data).

  For LUA applications, **startd** is 4 (three bytes of padding between the **trpad** field and the start of the data) in the first element of a message and 13 (12 bytes of padding) in subsequent elements.

  The extra bytes are used by the local node for additional header information; this avoids having to copy data into a new buffer when adding this information.

- Because **startd** indicates the index into **dataru** starting from 1, not 0, the first byte of valid data will always be at **dataru[startd–1]**.
- If **startd** is greater than **endd**, there is no valid data in the message.
- All fields within **dataru** are of type **CHAR**, except where the remarks indicate otherwise.

Note that where a buffer element has a **startd** of 1, 10, or 13, this only applies to the initial element in the chain of elements, and subsequent elements in the chain have a **startd** of 1. Messages with two distinct linked element chains in the message formats (for example Open(PLU) Request and Open(PLU) OK Response), have the **startd** field in the elements at the start of the chains as the value (1, 10, or 13) given in the message format, and the **startd** fields in all other elements as 1.

Open(SSCP)

The **Open(SSCP)** message is used by the application to open the SSCP connection. The Open request is sent by the application to the node, and the Open response comes from the node to the application.

Open(SSCP) Request

The **Open(SSCP) Request** message flows from the application to the node. It is used with an SSCP connection.

```
struct Open(SSCP) Request {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdrept;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER   ophdr.opresid;
    INTEGER   ophdr.icreditr;
    INTEGER   ophdr.icredits;
    CHAR      ophdr.opninfo1;
    CHAR      ophdr.opnpad1;
};
```

Element 1

```
struct Open(SSCP) Request {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Element 2

```
struct Open(SSCP) Request {
    PTRBFELT  hdreptr->elteptr->elteptr;
    INTEGER   hdreptr->elteptr->startd;
    INTEGER   hdreptr->elteptr->endd;
    CHAR      hdreptr->elteptr->trpad;
    CHAR[268] hdreptr->elteptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.

hdrept
Pointer to first buffer element.

numelts
Number of buffer elements (0x02).

msgtype
Message type OPENMSG (0x01).

srcl
Source locality.

srcp
Source partner (see Remarks).

srci
Source index.

destl
Destination locality.
destp
Destination partner.
desti
Destination index.
ophdr.openqual
Open qualifier REQU (0x01).
ophdr.opentype
Open type SSCPSEC (0x01).
ophdr.appltype
Application program interface type.

Possible values are:

FMI without chunking (0x02)

FMI with chunking (0x82) (see Remarks).

ophdr.opluno
Logical unit number (see Remarks).
ophdr.opresid
Resource identifier.
ophdr.icreditr
Reserved.
ophdr.icredits
Reserved.
ophdr.opninfo1
Reserved.
ophdr.opnpad1
Open force type (see Remarks).

Values can be:

OPEN_TEST (0x00)

OPEN_FORCE (0x01)

Element 1

hdreptr–>elteptr
Pointer to next buffer element.
hdreptr–>startd
Start of data in this buffer element (1).
hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved (1 byte).
hdreptr–>dataru
Data RU

The bits of this member represent the following values:

dataru[0–9] Source name.

Should be filled with blanks.

dataru[10–19] Destination name.

Set to the LU you want to communicate with.

dataru[20] Sense 4003 flag.

dataru[21] Sense 4004 flag.

dataru[22] Sense 4006 flag.

dataru[23] Sense 4007 flag.

dataru[24] Sense 4009 flag.

dataru[25] Sense 400A flag.

dataru[26] Sense 400B flag.

dataru[27] Sense 400C flag.

dataru[28] Sense 400D flag.

dataru[29] Sense 400F flag.

dataru[30] Sense 4011 flag.

dataru[31] Sense 4012 flag.

dataru[32] Sense 4014 flag.

dataru[33] High priority indicator.

Possible values are:

HIGH (0x01)

LOW (0x02)

dataru[34] LUA supported indicator.

Possible values are:

Supported (0x01)

Not supported (0x00)

dataru[35–36] Chunk size obtained from DMOD (see Remarks).

dataru[37] Segment delivery option.

Possible values are:

Do not deliver RU segments (0x00)

Deliver RU segments (0x01)

dataru[38] HLLAPI session identifier (see Remarks).

Element 2

hdreptr–>elteptr–>elteptr
Pointer to next buffer element (NIL).
hdreptr–>elteptr–>startd
Start of data in this buffer element (1).
hdreptr–>elteptr–>endd
End of data in this buffer element.
hdreptr–>elteptr–>trpad
Reserved.
hdreptr–>elteptr–>dataru
Data RU, as follows:

dataru[0]

ASCII string identifying the 3270 emulator (see Remarks).

Remarks

- The **Open(SSCP) Request** message consists of a buffer header and two buffer elements.
- The source L value, the destination LPI values, and the source name are reserved.
- For a 3270 emulator, the source P value must be set to S3PROD (0x12), which identifies the application as a 3270 emulator. The destination name should be set to the LU name or pool name taken from the 3270 user record (right-filled with ASCII spaces if fewer than 10 characters).
- An LUA application uses the source P value LUAPROD (0x1D). This is independent of the value of the LUA supported indicator (see below), which selects the LUA variant of the FMI.
- The SNS4003 to SNS4014 fields together with the high priority indicator are referred to in the text (see Opening the SSCP Connection) as the SSCP connection information control block (CICB). A value of 0x00 indicates that the

DFC receive check corresponding to the sense code is not supported for this LU. A value of 0x01 indicates that it is supported. Note that the corresponding send checks are always performed regardless of these values.

- The LU number is only used internally in the local node on the **Open(SSCP) Request**. It is generated from the destination name in the first element.
- The open force type field is used when locating resources across more than one server and for automatic activation of connections when the application wishes to use an LU for which the connection is inactive. The application does not need to set this flag; it is used by the DL-BASE. See Opening the SSCP Connection for details.
- The application program interface type field defines whether RU chunking is used from the local node to the application; this may be necessary if large RUs are being used. See Pacing and Chunking for more information on FMI chunking. This feature is not yet supported in Host Integration Server or SNA Server but is described here for completeness because support is planned for a future version.
- The chunk size field (at **dataru[35]**) is an integer value.
- The segment delivery option specifies whether the local node should deliver segments of RUs to the application as soon as they are received or should assemble whole RUs before delivering them to the application. Segment delivery allows the application to update the user's screen as soon as data is received ("window shading"), which can result in a faster perceived response. See Segment Delivery for more information. This option is required only when chunking is being used; it is included on this message so that the local node can calculate the initial chunk credit values on the corresponding PLU connection. The option must still be set on the **Open(PLU) Response**; the setting specified on that message will override the one specified here if there is a conflict. If this happens, the initial credit values may not be suitable.
- The LUA supported indicator specifies whether the application uses the LUA variant of the FMI.
- If the element is shorter than (s+34) bytes, Host Integration Server or SNA Server assumes no LUA and no chunking. This ensures backward compatibility with previous versions of the local node software in which these options were not available.
- The HLLAPI session identifier is a single ASCII character that identifies the 3270 display session to which the **Open(SSCP)** applies. HLLAPI uses this to identify a particular 3270 presentation space to which an HLLAPI function refers; it is also referred to by 3270 as the session's short name, or by HLLAPI as the presentation space identifier (PS identifier). If the 3270 emulator does not support session identifiers, this field should be set to zero.
- The second element contains an ASCII string that you can use to identify the type of 3270 emulator, such as "Select SNA Server OS/2 3270." This string will be logged in the audit log file by the client's DL-BASE and can also be seen in traces. The **startd** and **endd** fields must be set up to define the limits of this string.

Open(SSCP) Response

The **Open(SSCP) Response** message flows from the node to the application. It is used with an SSCP connection.

```
struct Open(SSCP) Response {
    PTRBFHDR  nxtqptr;
    PTRBFELT hdreptr;
    CHAR     numelts;
    CHAR     msgtype;
    CHAR     srcl;
    CHAR     srcp;
    INTEGER  srci;
    CHAR     destl;
    CHAR     destp;
    INTEGER  desti;
    CHAR     ophdr.openqual;
    CHAR     ophdr.opentype;
    CHAR     ophdr.appltype;
    CHAR     ophdr.opluno;
    INTEGER  ophdr.opresid;
    INTEGER  ophdr.operr1;
    INTEGER  ophdr.operr2;
};
```

Element 1

```
struct Open(SSCP) Response {
    PTRBFELT  hdreptr->elteptr;
    INTEGER   hdreptr->startd;
    INTEGER   hdreptr->endd;
    CHAR      hdreptr->trpad;
    CHAR[256] dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element.
numelts
Number of buffer elements (0x01).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
ophdr.openqual
Open qualifier.

Possible values are:

RSPOK (0x02)
RSPERR (0x03)

ophdr.opentype
Open type SSCPSEC (0x01).
ophdr.appltype
Application program interface type.

Possible values are:

0x02 (FMI application)

ophdr.opluno
Logical unit number.
ophdr.opresid
Resource identifier.
ophdr.operr1
Error code 1.
ophdr.operr2
Error code 2.

Element 1

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element (1).
hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved (1 byte).
hdreptr–>dataru
Data RU

Bits in this member have the following values:

dataru[0–9] Source name.

dataru[10–19] Destination name.

dataru[20–27] Name of the local node that accepted the Open.

dataru[28–35] Name of the connection used by the LU.

dataru[36–37] The local node's internal identifier for the connection (see Remarks).

dataru[38] The type of link service used by the connection:

CESLINK (03) - SDLC
CESX25 (04) - X.25
CESDFT (10) - DFT
CESTR (11) - Token Ring
CESTCPIP (30) - TCP/IP
CESRELAY (31) - Frame Relay
CESCHANL (32) - Channel
CESISDN (33) - ISDN
CESETHER (34) - Ethernet 802.2

Remarks

- The **Open(SSCP) Response** message consists of a buffer header and a single buffer element.
- If the open qualifier is RSPERR, the error code is valid (see Error and Sense Codes), and the LPIs and names are undefined.
- The LU number indicates the LU selected by the local node from the configuration data (see Opening the SSCP Connection.
- When the **Open(SSCP)** is for an LU group, the source name contains the name of the selected LU.
- The connection identifier is an integer value. It uniquely identifies a particular connection on this local node; all sessions using the same connection will return the same identifier. This value is typically used when a link error is received on one session to determine which other sessions will be affected.

Open(PLU)

The **Open(PLU)** message is used by the local node to open the PLU connection with the application on receipt of a BIND command from the host.

Open(PLU) Request

The **Open(PLU) Request** message flows from the node to the application. It is used with a PLU connection.

```
struct Open(PLU) Request {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER   ophdr.opresid;
    INTEGER   ophdr.icreditr;
    INTEGER   ophdr.icredits;
    CHAR      ophdr.opninfo1;
};
```

Element 1

```
struct Open(PLU) Request {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Element 2

```
struct Open(PLU) Request {
    PTRBFELT  hdreptr->elteptr->elteptr;
    INTEGER   hdreptr->elteptr->startd;
    INTEGER   hdreptr->elteptr->endd;
    CHAR      hdreptr->elteptr->trpad;
    CHAR[ ]   hdreptr->elteptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element.
numelts
Number of buffer elements (0x02).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl

Destination locality.

destp
Destination partner.

desti
Destination index.

ophdr.openqual
Open qualifier REQU (0x01).

ophdr.opentype
Open type LUSEC (0x02).

ophdr.appltype
Application program interface type.

Possible values are:

0x02 (FMI application)

ophdr.opluno
Logical unit number.

ophdr.opresid
Resource identifier.

ophdr.icreditr
Initial credit for flow from application to local node: zero (no flow control).

ophdr.icredits
Recommended initial credit for flow from local node to application:
Pacing window + 1.

ophdr.opninfo1
Negotiable bind indicator

Possible values are:

Bind is not negotiable (0x00)

Bind is negotiable (0x01)

Element 1

hdreptr–>elteptr
Pointer to buffer element.

hdreptr–>startd
Start of data in this buffer element (1).

hdreptr–>endd
End of data in this buffer element.

hdreptr–>trpad
Reserved.

hdreptr–>dataru
Data RU, as follows:

dataru[0–9] Source name.

dataru[10–19] Destination name.

dataru[20] Secondary pacing send window.

dataru[21] Secondary pacing receive window.

dataru[22–23] Secondary send maximum RU size (see Remarks).

dataru[24–25] Primary send maximum RU size (see Remarks).

dataru[26] Secondary send chunk size (in units of elements).

dataru[27] Primary send chunk size (in units of elements).

Element 2

hdreptr–>elteptr–>elteptr
Pointer to buffer element (NIL).

hdreptr–>elteptr–>startd
Start of data in this buffer element (13).

hdreptr–>elteptr–>endd

End of data in this buffer element.

hdreptr–>elteptr–>trpad

Reserved.

hdreptr–>elteptr–>dataru

Data RU, as follows:

dataru[13] The BIND RU received from the host.

Remarks

- The **Open(PLU) Request** message consists of a buffer header, an initial element containing the source and destination names, RU sizes, and so on, followed by a second element containing the BIND RU received from the host.
- The source LPI and the L and P parts of the destination LPI are valid, but the I part of the destination LPI is reserved.
- The two send maximum RU size fields (in **dataru[22–25]**) are both integer values.
- The BIND RU can be up to 256 bytes in length.
- If the application is using the LUA variant of the FMI (see FMI Concepts), the BIND RU is preceded by its TH and RH; the **startd** field of the second element points to the TH.
- The LU number matches that allocated to the named application on the Open(SSCP) Response.
- The resource identifier matches the value used by the application on the Open(SSCP) Request.
- If chunking was specified on the **Open(SSCP) Request**, the **icredits** (initial credit from local node to application) field specifies the number of chunks, rather than RUs, that can be transmitted. The two send chunk size parameters are specified in units of elements (each element contains up to 256 bytes of RU data). A value of zero indicates that the chunk size is not the limiting factor in determining the size of messages; the limiting factor is the RU size or the segment size, so chunking is not required. In this case, credit will still be used, with the unit of credit being a message.
- The **icreditr** (initial credit from application to local node) field is not used and must be set to zero.

Open(PLU) OK Response

The **Open(PLU) OK Response** message flows from the application to the node. It is used with a PLU connection.

```
struct Open(PLU) OK Response {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER   ophdr.opresid;
    INTEGER   ophdr.icreditr;
    INTEGER   ophdr.icredits;
    CHAR      ophdr.opninfo1;
};
```

Element 1

```
struct Open(PLU) OK Response {
    PTRBFELT hdreptr->elteptr;
    INTEGER hdreptr->startd;
    INTEGER hdreptr->endd;
    CHAR hdreptr->trpad;
    CHAR[268] hdreptr->dataru;
};
```

Element 2

```
struct Open(PLU) OK Response {
    PTRBFELT   hdreptr->elteptr->elteptr;
    INTEGER    hdreptr->elteptr->startd;
    INTEGER    hdreptr->elteptr->endd;
    CHAR       hdreptr->elteptr->trpad;
    CHAR[268]  hdreptr->elteptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element.
numelts
Number of buffer elements (0x02).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl

Destination locality.

destp
Destination partner.

desti
Destination index.

ophdr.openqual
Open qualifier RSPOK (0x02).

ophdr.opentype
Open type LUSEC (0x02).

ophdr.appltype
Application program interface type.

Possible values are:

0x02 (FMI application)

ophdr.opluno
Logical unit number.

ophdr.opresid
Resource identifier.

ophdr.icreditr
Initial credit for flow from application to local node: zero.

ophdr.icredits
Initial credit for flow from local node to application; only valid if APPLPAC = 0x01.

ophdr.opninfo1
Negotiable bind indicator.

Possible values are:

Bind is not negotiable (0x00)

Bind is negotiable (0x01)

Element 1

hdreptr–>elteptr
Pointer to buffer element.

hdreptr–>startd
Start of data in this buffer element (1).

hdreptr–>endd
End of data in this buffer element.

hdreptr–>trpad
Reserved.

hdreptr–>dataru
Data RU, as follows:

dataru[0–9] Source name.

dataru[10–19] Destination name.

dataru[20] Segment delivery option.

Possible values are:

Do not deliver RU segments (0x00)

Deliver RU segments (0x01)

dataru[21] Application pacing option.

Possible values are:

No application pacing (0x00)

Application pacing (0x01)

dataru[22] Application cancel option

Cancel is generated by:

local node (0x00)

application (0x01)

dataru[23] Application transaction numbers option

Transaction numbers are:

not supported by application (0x00)

supported by application (0x01)

dataru[24] BIND table index

Possible values are:

BIND_TABLE_INDEX_PRT (1) (printer session)

BIND_TABLE_INDEX_CRT (2) (display session)

Element 2

hdreptr–>elteptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>elteptr–>startd
Start of data in this buffer element (13).
hdreptr–>elteptr–>endd
End of data in this buffer element.
hdreptr–>elteptr–>trpad
Reserved.
hdreptr–>elteptr–>dataru
Data RU, as follows:

dataru[13] The BIND RU.

Remarks

- The **Open(PLU) OK Response** message consists of a buffer header, an initial element containing the source and destination names and CICB, followed by elements containing the BIND RU received from the host.
- The application should reflect the source and destination LPIs and the source and destination names from the Open(PLU) Request and must supply the I part of the source LPI.

The fields from segment delivery option to bind table index (in the first element) are referred to in the text as the PLU connection information control block (CICB). See Opening the PLU Connection for further information on the contents of the CICB.

- The BIND RU can be up to 256 bytes in length.
- For LUA, the BIND RU is not preceded by its TH and RH (contrast with the Open(PLU) Request, where the TH and RH are included).
- As in the **Open(PLU) Request**, the **icredits** value is in units of chunks if chunking is being used.

Open(PLU) Error Response

The **Open(PLU) Error Response** message flows from the application to the node. It is used with a PLU connection.

```
struct Open(PLU) Error Response {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER  ophdr.opresid;
    INTEGER  ophdr.operr1;
    INTEGER  ophdr.operr2;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
ophdr.openqual
Open qualifier RSPERR (0x03).
ophdr.opentype
Open type LUSEC (0x02).
ophdr.appltype
Application program interface type.

Possible values are:

0x02 (FMI application)

ophdr.opluno
Logical unit number.
ophdr.opresid
Resource identifier.
ophdr.operr1
Error code 1.
ophdr.operr2

Error code 2.

Remarks

- The **Open(PLU) Error Response** message consists of a buffer header only.
- The application should reflect the source and destination LPIs.

Open(PLU) OK Confirm

The **Open(PLU) OK Confirm** message flows from the node to the application. It is used with a PLU connection.

```
struct Open(PLU) OK Confirm {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   dsti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER   ophdr.opresid;
    INTEGER   ophdr.icreditr;
    INTEGER   ophdr.icredits;
    CHAR      ophdr.opninfo1;
};
```

Element

```
struct Open(PLU) OK Confirm {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element.
numelts
Number of buffer elements (0x01).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
dsti
Destination index.
ophdr.openqual
Open qualifier CONFOK (0x04).
ophdr.opentype
Open type LUSEC (0x02).
ophdr.appltype

Application program interface type.

0x02 (FMI application)

ophdr.opluno
Logical unit number.
ophdr.opresid
Resource identifier.
ophdr.icreditr
Reserved.
ophdr.icredits
Reserved.
ophdr.opninfo1
PLU address.

Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element (1).
hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU, as follows:

dataru[0] FM profile.

dataru[1] TS profile.

dataru[2] Primary chaining use.

dataru[3] Primary request control mode.

dataru[4] Primary chain response protocol.

dataru[5] Primary two-phase commit.

dataru[6] Primary compression indicator.

dataru[7] Primary send end bracket (EB) indicator.

dataru[8] Secondary chaining use.

dataru[9] Secondary request control mode.

dataru[10] Secondary chain response protocol.

dataru[11] Secondary two-phase commit.

dataru[12] Secondary compression indicator.

dataru[13] Secondary send EB indicator.

dataru[14] FM header usage.

dataru[15] Bracket usage.

Possible values are:

Brackets not used (0x00)

Brackets used (0x01)

dataru[16] Bracket reset state.

Possible values are:

Bracket reset state between-brackets (BETB) (0x01)

Bracket reset state in-bracket (INB) (0x02)

dataru[17] Bracket termination rule.

dataru[18] Alternate code set indicator.

dataru[19] Sequence number availability.

dataru[20] Normal-flow send/receive mode.

dataru[21] Half-duplex flip-flop reset.

dataru[22] Secondary pacing send window.

dataru[23] Secondary pacing receive window.

dataru[24–25] Secondary send maximum RU size (INTEGER value).

dataru[26–27] Primary send maximum RU size (INTEGER value).

dataru[28] LU-LU session type.

dataru[29] PLU name size.

dataru[30–37] PLU name (EBCDIC).

dataru[38] Session type 1: PS FMH type.

dataru[39] PS data stream profile.

dataru[40] Number of outstanding destinations.

dataru[41] Compacted data indicator.

dataru[42] PDIR allowed indicator.

dataru[43] Session type 2 or 3: query support.

dataru[44] Dynamic screen size.

dataru[45] Basic row size.

dataru[46] Basic column size.

dataru[47] Alternate row size.

dataru[48] Alternate column size.

Remarks

- The **Open(PLU) OK Confirm** message consists of a buffer header and one element.
- The message does not carry source and destination names; both LPIs are valid.
- The contents of **dataru** are referred to in the text as the PLU bind information control block (BICB). The BICB is only valid for an open-qualifier of CONFOK. See Opening the PLU Connection for further information on the contents of the BICB.

Open(PLU) Error Confirm

The **Open(PLU) Error Confirm** message flows from the node to the application. It is used with a PLU connection.

```
struct Open(PLU) Error Confirm {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      ophdr.openqual;
    CHAR      ophdr.opentype;
    CHAR      ophdr.appltype;
    CHAR      ophdr.opluno;
    INTEGER   ophdr.opresid;
    INTEGER   ophdr.operr1;
    INTEGER   ophdr.operr2;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to first buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type OPENMSG (0x01).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
ophdr.openqual
Open qualifier CONFERR (0x05).
ophdr.opentype
Open type LUSEC (0x02).
ophdr.appltype
Application program interface type

Possible values are:

0x02 (FMI application)

ophdr.opluno
Logical unit number.
ophdr.opresid
Resource identifier.
ophdr.operr1
Error code 1.
ophdr.operr2

Error code 2.

Remarks

- The **Open(PLU) Error Confirm** message consists of a buffer header only.
- The error codes are valid (see Error and Sense Codes. An **Open(PLU) Error Confirm** closes the connection.

Close(SSCP)

The **Close(SSCP)** message closes an open SSCP connection.

Close(SSCP) Request

The **Close(SSCP) Request** message flows from the application to the node. It is used with an SSCP connection.

```
struct Close(SSCP) Request {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   destr;
    CHAR      clhdr.closqual;
    CHAR      clhdr.clstype;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type CLOSEMSG (0x02).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
clhdr.closqual
Close qualifier REQU (0x01).
clhdr.clstype
Close subtype SSCPSEC (0x01).

Remarks

- The **Close(SSCP) Request** message consists of a buffer header only; there is no buffer element.

Close(SSCP) Response

The **Close(SSCP) Response** message flows from the node to the application. It is used with an SSCP connection.

```
struct Close(SSCP) Response {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      clhdr.closqual;
    CHAR      clhdr.clstype;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type CLOSEMSG (0x02).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
clhdr.closqual
Close qualifier RSPOK (0x02).
clhdr.clstype
Close subtype SSCPSEC (0x01).

Remarks

- The **Close(SSCP) Response** message consists of a buffer header only; there is no buffer element.
- The **Close(SSCP)** protocol is unconditional. It is not possible for the local node to keep the SSCP connection open after the application sends **Close(SSCP)**.

Close(PLU)

The **Close(PLU)** message closes an open PLU connection.

Close(PLU) Request

The **Close(PLU) Request** message flows from the node to the application and from the application to the node. It is used with a PLU connection.

```
struct Close (PLU) Request {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      clhdr.closqual;
    CHAR      clhdr.clstype;
    CHAR      clhdr.clsctl;
    CHAR      clhdr.clspad1;
    INTEGER   clhdr.clspad2;
    INTEGER   clhdr.clserr1;
};
```

LUA only (see Remarks):
Element

```
struct Close (PLU) Request {
    PTRBFELT hdreptr->elteptr;
    INTEGER hdreptr->startd;
    INTEGER hdreptr->endd;
    CHAR hdreptr->trpad;
    CHAR[268] hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type CLOSEMSG (0x02).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
clhdr.closqual
Close qualifier REQU (0x01).
clhdr.clstype
Close subtype LUSEC (0x02).
clhdr.clsctl

Close control

Possible values are:

CLNORMAL (0x01) - normal
CLBIND (0x02) - bind forthcoming
CLCFAERR (0x03) - CFA error
CLPUINAC (0x04) - PU inactive
CLLUINAC (0x05) - LU inactive
CLLNKERR (0x06) - link error
CLBFSHRT (0x07) - node buffer shortage
CLRCVCHK (0x08) - DFC receive check
CLSLUTRM (0x09) - SLU termination

clhdr.clspad1
Reserved.
clhdr.clspad2
Reserved.
clhdr.clserr1
Error code (only valid for close control = link error).

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element (13).
hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
The UNBIND RU received from the host, with its TH and RH.

Remarks

- If the application is using the LUA variant of the FMI (see FMI Concepts), and the **Close(PLU) Request** was generated by receipt of an UNBIND from the host, then the element is included and **startd** points to the TH of the UNBIND message.
- In all other cases (for example, if the **Close(PLU) Request** was generated by the local node as a result of a link outage), the message consists of a buffer header only; there is no buffer element.
- The close control field is only valid on messages from the local node to the application.
- If the close control field specifies link error, then the error code field gives the link outage code.

Close(PLU) Response

The **Close(PLU) Response** message flows from the node to the application and from the application to the node. It is used with a PLU connection.

```
struct Close(PLU) Response {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      clhdr.closqual;
    CHAR      clhdr.clstype;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type CLOSEMSG (0x02).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
clhdr.closqual
Close qualifier RSPOK (0x02).
clhdr.clstype
Close subtype LUSEC (0x02).

Remarks

- The **Close(PLU) Response** message consists of a buffer header only; there is no buffer element.
- The **Close(PLU)** protocol is unconditional. It is not possible for the recipient of a Close(PLU) Request (either the local node or an application) to keep the PLU connection open; the only valid response is **Close(PLU) Response** with the close qualifier as RSPOK.

Data

Data messages carry both inbound and outbound data between the application and the local node on all three connections. See Data Flow for a detailed description of outbound and inbound data flows.

The **Data** message flows from the node to the application and from the application to the node. It is used with both the SSCP and the PLU connections.

```
struct Data {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      dfhdr.fhackrqd;
    CHAR      dfhdr.fhpad1;
    INTEGER   dfhdr.fhmsgkey;
    CHAR      dfhdr.fhflags1;
    CHAR      dfhdr.fhflags2;
    INTEGER   dfhdr.fhpad2;
    INTEGER   dfhdr.fhpad3;
    INTEGER   dfhdr.fhseqno;
};
```

Element

```
struct Data {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element.
numelts
Number of buffer elements.
msgtype
Message type DATAFMI (0x20).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
dfhdr.fhackrqd
Acknowledgment required indicator.

Possible values are:

NOACKREQ (0x00)
ACKREQ (0x01)

dfhdr.fhpad1
Reserved.
dfhdr.fhmsgkey
Message key.
dfhdr.fhflags1
Application flag 1.
dfhdr.fhflags2
Application flag 2.
dfhdr.fhpad2
Reserved.
dfhdr.fhpad3
Reserved.
dfhdr.fhseqno
Sequence number.

Element

hdreptr–>elteptr
Pointer to buffer element.
hdreptr–>startd
Start of data in this buffer element.

Possible values are:

Non-LUA:

13, or 10 for second and subsequent segments of outbound segmented RUs.
LUA, inbound data:

4 in first element, 13 in subsequent elements.

hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU.

Remarks

- The use of the acknowledgment required indicator in both inbound and outbound data acknowledgment protocols is described in Data Flow.

The use of the application flag fields is described in Application Flags (see note below for LUA).

- The sequence number is undefined for inbound data but contains the corresponding SNA sequence number for outbound data.
- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH and (if appropriate) RH are included in the data, and the **startd** field points to the TH. The **fhmsgkey**, **fhflags1**, **fhflags2**, and **fhseqno** fields are undefined and should not be used; the corresponding data from the element should be used instead.

Status-Acknowledge

Status-Acknowledge messages flow between the application and the local node as part of the outbound and inbound data acknowledgment protocols. See Data Flow for a detailed description of outbound and inbound acknowledgment protocols.

> **Note** The format of this message is slightly different for messages from the application to the local node on the PLU connection, as is explained in the following topic.

Status-Acknowledge(Ack)

The **Status-Acknowledge(Ack)** message flows from the node to the application and from the application to the node, and is used with both SSCP and PLU connections.

The following structure shows the message format for all SSCP messages and for PLU messages flowing from the node to the application.

```
struct Status-Acknowledge(Ack) {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stackhdr.akstat;
    CHAR      sfhdr.stackhdr.akqual;
    INTEGER   sfhdr.stackhdr.akmsgkey;
    CHAR      sfhdr.stackhdr.akflags1;
    CHAR      sfhdr.stackhdr.akflags2;
    INTEGER   sfhdr.stackhdr.aknumb1;
    INTEGER   sfhdr.stackhdr.aknumb2;
    INTEGER   sfhdr.stackhdr.akseqno;
};
```

LUA only (see Remarks):
Element

```
struct Status-Acknowledge(Ack) {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stackhdr.akstat

Status type ACK (0x01).

sfhdr.stackhdr.akqual
Acknowledgment type ACKPOS (0x02).

sfhdr.stackhdr.akmsgkey
Message key.

sfhdr.stackhdr.akflags1
Application flag 1.

sfhdr.stackhdr.akflags2
Application flag 2.

sfhdr.stackhdr.aknumb1
Undefined.

sfhdr.stackhdr.aknumb2
Reserved.

sfhdr.stackhdr.akseqno
SNA sequence number.

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).

hdreptr–>startd
Start of data in this buffer element.

13 or 10 for second and subsequent segments of outbound segmented RUs

hdreptr–>endd
End of data in this buffer element.

hdreptr–>trpad
Reserved.

hdreptr–>dataru
Data RU.

The message format for PLU messages flowing from the application to the node is identical to the preceding format, except that the application flag 1 and application flag 2 fields are not used. They are replaced by the following INTEGER field:

sfhdr.stackhdr.akmsgtim
Last host response time

Possible values are:

0xFFFF - no response time measured
0x*nnnn* - last response time measured, in units of 0.1 second

INTEGER	*sfhdr.stackhdr.akmsgtim*	Last host response time 0xFFFF - no response time measured 0x*nnnn* - last response time measured, in units of 0.1 second

Remarks

- The message key and application flags reflect the message key and application flags of the data message to which this is an acknowledgment (see note on LUA below).
- For outbound **Status-Acknowledge(Ack)** messages from the local node to the application, the SNA sequence number gives the sequence number of the inbound data message to which this is an acknowledgment (see note on LUA below); it is normally used only by TS profile 4 applications.
- For inbound **Status-Acknowledge(Ack)** messages from the application to the local node, the SNA sequence number reflects the sequence number of the outbound data message to which this is an acknowledgment.
- If the host specified that response time statistics are to be maintained (see RTM Parameters and Response Time Monitor Data for details), the application is responsible for measuring and reporting response times to the local node, using the **akmsgtim** field of this message.
- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH and (if appropriate) RH are included in the data, and the **startd** field points to the TH. The **akmsgkey**, **akflags1**, and **akflags2** fields are undefined and should not be

used; the corresponding data from the element should be used instead. The **akseqno** field is similarly undefined on messages from the local node to the application; it must be set on messages from the application to the local node. The **akseqno** field is used to hold the sequence number of the request being acknowledged.

- If the application is not using the LUA variant of the FMI, the message consists of a buffer header only; there is no buffer element.

Status-Acknowledge(Nack-1)

The **Status-Acknowledge(Nack-1)** message flows from the node to the application and from the application to the node. It is used with both SSCP and PLU connections.

```
struct Status-Acknowledge(Nack-1) {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stackhdr.akstat;
    CHAR      sfhdr.stackhdr.akqual;
    INTEGER   sfhdr.stackhdr.akmsgkey;
    CHAR      sfhdr.stackhdr.akflags1;
    CHAR      sfhdr.stackhdr.akflags2;
    INTEGER   sfhdr.stackhdr.aknumb1;
    INTEGER   sfhdr.stackhdr.aknumb2;
    INTEGER   sfhdr.stackhdr.akseqno;
};
```

LUA only (see Remarks):
Element

```
struct Status-Acknowledge(Nack-1) {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stackhdr.akstat
Status type ACK (0x01).

sfhdr.stackhdr.akqual
Acknowledgment type ACKNEG1 (0x03).
sfhdr.stackhdr.akmsgkey
Message key.
sfhdr.stackhdr.akflags1
Application flag 1.
sfhdr.stackhdr.akflags2
Application flag 2.
sfhdr.stackhdr.aknumb1
Sense data 1.
sfhdr.stackhdr.aknumb2
Sense data 2.
sfhdr.stackhdr.akseqno
SNA sequence number.

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element.

13 or 10 for second and subsequent segments of outbound segmented RUs

hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU.

Remarks

- The message key and application flags reflect the message key and application flags of the data message to which this is a negative acknowledgment (see note on LUA below).
- For **Status-Acknowledge(Nack-1)** messages from the local node to the application, the sense data reflects the sense data in the SNA negative response.
- For **Status-Acknowledge(Nack-1)** messages from the application to the local node, the sense data fields are those intended for the SNA negative response to the host.
- For outbound **Status-Acknowledge(Nack-1)** messages from the local node to the application, the SNA sequence number gives the sequence number of the inbound data message to which this is a negative acknowledgment (see note on LUA below).
- For inbound **Status-Acknowledge(Nack-1)** messages from the application to the local node, the SNA sequence number reflects the sequence number of the outbound data message to which this is a negative acknowledgment.
- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH and (if appropriate) RH are included in the data, and the **startd** field points to the TH. The **akmsgkey**, **akflags1**, and **akflags2** fields are undefined and should not be used; the corresponding data from the element should be used instead. The **akseqno** field is similarly undefined on messages from the local node to the application; it must be set on messages from the application to the local node.
- If the application is not using the LUA variant of the FMI, the message consists of a buffer header only; there is no buffer element.

Status-Acknowledge(Nack-2)

The **Status-Acknowledge(Nack-2)** message flows from the node to the application. It is used with both SSCP and PLU connections.

```
struct Status-Acknowledge(Nack-2) {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stackhdr.akstat;
    CHAR      sfhdr.stackhdr.akqual;
    INTEGER   sfhdr.stackhdr.akmsgkey;
    CHAR      sfhdr.stackhdr.akflags1;
    CHAR sfhdr.stackhdr.akflags2;
    INTEGER sfhdr.stackhdr.aknumb1;
    INTEGER sfhdr.stackhdr.aknumb2;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stackhdr.akstat
Status type ACK (0x01).
sfhdr.stackhdr.akqual
Acknowledgment type ACKNEG2 (0x04).
sfhdr.stackhdr.akmsgkey
Message key.
sfhdr.stackhdr.akflags1
Reserved.
sfhdr.stackhdr.akflags2
Critical failure indicator.

Possible values are:

Noncritical failure (0x00)
Critical failure (0x01)

sfhdr.stackhdr.aknumb1

Error code 1.

sfhdr.stackhdr.aknumb2

Error code 2.

Remarks

- The **Status-Acknowledge(Nack-2)** message consists of a buffer header only; there is no buffer element.
- The message key refers to the message key in the inbound data message to which this is a negative acknowledgment.
- See Error and Sense Codes for error codes.

Status-Acknowledge(ACKLUA)

The **Status-Acknowledge(ACKLUA)** message is for LUA applications only. It flows from the node to the application, and is used with both the SSCP and PLU connections.

```
struct Status-Acknowledge(ACKLUA) {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stackhdr.akstat;
    CHAR      sfhdr.stackhdr.akqual;
    INTEGER   sfhdr.stackhdr.akmsgkey;
    CHAR      sfhdr.stackhdr.akflags1;
    CHAR      sfhdr.stackhdr.akflags2;
    INTEGER   sfhdr.stackhdr.aknumb1;
    INTEGER   sfhdr.stackhdr.aknumb2;
    INTEGER   sfhdr.stackhdr.akseqno;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stackhdr.akstat
Status type ACK (0x01).
sfhdr.stackhdr.akqual
Acknowledgment type.
sfhdr.stackhdr.akmsgkey
Message key.
sfhdr.stackhdr.akflags1
Application flag 1.
sfhdr.stackhdr.akflags2
Application flag 2.
sfhdr.stackhdr.aknumb1
Number of replies.
sfhdr.stackhdr.aknumb2
Reserved.

sfhdr.stackhdr.akseqno
SNA sequence number.

Remarks

- The message key and application flags are undefined and should not be checked.
- The SNA sequence number gives the sequence number of the inbound data message to which this is an acknowledgment.

Status-Control

See Status-Control Message for details of **Status-Control** message usage and for a summary of control type codes.

Status-Control(...) Request

The **Status-Control(...) Request** message flows from the node to the application and from the application to the node. It is used with a PLU connection.

```
struct Status-Control(...) Request {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stctlhdr.ctlstat;
    CHAR      sfhdr.stctlhdr.ctlqual;
    CHAR      sfhdr.stctlthdr.ctltype
    CHAR      sfhdr.stctlhdr.ctlack;
    CHAR      sfhdr.stctlhdr.ctlflag1;
    CHAR      sfhdr.stctlhdr.ctlflag2;
    INTEGER   sfhdr.stctlhdr.ctlnumb1;
    INTEGER   sfhdr.stctlhdr.ctlnumb2;
    INTEGER   sfhdr.stctlhdr.ctlmsgk;
};
```

LUA only (see Remarks):
Element

```
struct Status-Control(...) Request {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stctlhdr.ctlstat
Status type STCNTRL (0x02).

sfhdr.stctlhdr.ctlqual
Control qualifier (0x01).
sfhdr.stctlhdr.ctltype
Control type.
sfhdr.stctlhdr.ctlack
Acknowledgment required indicator.

Possible values are:

No acknowledgment required (0x00)
Acknowledgment required (0x01)

sfhdr.stctlhdr.ctlflag1
Application flag 1.
sfhdr.stctlhdr.ctlflag2
Application flag 2 (see STSN).
sfhdr.stctlhdr.ctlnumb1
Code 1.
sfhdr.stctlhdr.ctlnumb2
Code 2.
sfhdr.stctlhdr.ctlmsgk
Message key.

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element.

Possible values are:

13 or 10 for second and subsequent segments of outbound segmented RUs

hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU.

Remarks

- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH, RH, and RU are included in the data element, and the **startd** field points to the TH. The **ctlflag1** and **ctlflag2** bytes are not defined and should not be used; the appropriate values from the data should be used instead.
- If the application is not using the LUA variant of the FMI, the message consists of a buffer header only; there is no buffer element.
- See the table in Status-Control Message for a summary of **Status-Control** control type codes.
- The code 1 and code 2 fields apply only for **Status-Control** LUSTAT, SIGNAL, and STSN messages.
- The application flag byte 2 is used for the **Status-Control** STSN control byte (see Recovery).

Status-Control(...) Acknowledge

The **Status-Control(...) Acknowledge** message flows from the node to the application and from the application to the node. It is used with a PLU connection.

```
struct Status-Control(...) Acknowledge {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stctlhdr.ctlstat;
    CHAR      sfhdr.stctlhdr.ctlqual;
    CHAR      sfhdr.stctlhdr.ctltype;
    CHAR      sfhdr.stctlhdr.ctlack;
    CHAR      sfhdr.stctlhdr.ctlflag1;
    CHAR      sfhdr.stctlhdr.ctlflag2;
    INTEGER   sfhdr.stctlhdr.ctlnumb1;
    INTEGER   sfhdr.stctlhdr.ctlnumb2;
    INTEGER   sfhdr.stctlhdr.ctlmsgk;
};
```

LUA only (see Remarks):
Element

```
struct Status-Control(...) Acknowledge {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stctlhdr.ctlstat
Status type STCNTRL (0x02).

sfhdr.stctlhdr.ctlqual
Control qualifier ACKPOS (0x02).
sfhdr.stctlhdr.ctltype
Control type.
sfhdr.stctlhdr.ctlack
Reserved.
sfhdr.stctlhdr.ctlflag1
Application flag 1.
sfhdr.stctlhdr.ctlflag2
Application flag 2 (see STSN).
sfhdr.stctlhdr.ctlnumb1
Code 1.
sfhdr.stctlhdr.ctlnumb2
Code 2.
sfhdr.stctlhdr.ctlmsgk
Message key.

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element.

Possible values are:

13 or 10 for second and subsequent segments of outbound segmented RUs

hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU.

Remarks

- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH, RH, and RU are included in the data element, and the **startd** field points to the TH. The **ctlflag1** and **ctlflag2** bytes are not defined and should not be used; the appropriate values from the data should be used instead.
- If the application is not using the LUA variant of the FMI, the message consists of a buffer header only; there is no buffer element.
- See the table in Status-Control Message for a summary of **Status-Control** control type codes.
- The code 1 and code 2 fields apply only for **Status-Control(STSN) Acknowledge** messages, where they are the application's secondary-to-primary and primary-to-secondary sequence numbers respectively.
- For messages from the application to the local node, the message key field must match the message key in the corresponding **Status-Control Request**.

Status-Control(...) Negative-Acknowledge-1

The **Status-Control(...) Negative-Acknowledge-1** message flows from the node to the application and from the application to the node. It is used with a PLU connection.

```
struct Status-Control(...) Negative-Acknowledge-1 {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stctlhdr.ctlstat;
    CHAR      sfhdr.stctlhdr.ctlqual;
    CHAR      sfhdr.stctlhdr.ctltype
    CHAR      sfhdr.stctlhdr.ctlack;
    CHAR      sfhdr.stctlhdr.ctlflag1;
    CHAR      sfhdr.stctlhdr.ctlflag2;
    INTEGER   sfhdr.stctlhdr.ctlnumb1;
    INTEGER   sfhdr.stctlhdr.ctlnumb2;
    INTEGER   sfhdr.stctlhdr.ctlmsgk;
};
```

LUA only (see Remarks):
Element

```
struct Status-Control(...) Negative-Acknowledge-1 {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL if not using LUA).
numelts
Number of buffer elements (0x00 if not using LUA).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stctlhdr.ctlstat
Status type STCNTRL (0x02).

sfhdr.stctlhdr.ctlqual
Control qualifier ACKNEG1 (0x03).
sfhdr.stctlhdr.ctltype
Control type.
sfhdr.stctlhdr.ctlack
Reserved.
sfhdr.stctlhdr.ctlflag1
Application flag 1.
sfhdr.stctlhdr.ctlflag2
Application flag 2.
sfhdr.stctlhdr.ctlnumb1
Sense code 1.
sfhdr.stctlhdr.ctlnumb2
Sense code 2.
sfhdr.stctlhdr.ctlmsgk
Message key.

LUA only (see Remarks):
Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this buffer element.

13 or 10 for second and subsequent segments of outbound segmented RUs

hdreptr–>endd
End of data in this buffer element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU.

Remarks

- If the application is using the LUA variant of the FMI (see FMI Concepts), the TH, RH, and RU are included in the data element, and the **startd** field points to the TH. The **ctlflag1** and **ctlflag2** bytes are not defined and should not be used; the appropriate values from the data should be used instead.
- If the application is not using the LUA variant of the FMI, the message consists of a buffer header only; there is no buffer element.
- See the table in Status-Control Message for a summary of **Status-Control** control type codes.
- For messages from the application to the local node, the message key field must match the message key in the corresponding **Status-Control** request.

Status-Control(...) Negative-Acknowledge-2

The **Status-Control(...) Negative-Acknowledge-2** message flows from the node to the application. It is used with a PLU connection.

```
struct Status-Control(...) Negative-Acknowledge-2 {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stctlhdr.ctlstat;
    CHAR      sfhdr.stctlhdr.ctlqual;
    CHAR      sfhdr.stctlhdr.ctltype
    CHAR      sfhdr.stctlhdr.ctlack;
    CHAR      sfhdr.stctlhdr.ctlflag1;
    CHAR      sfhdr.stctlhdr.ctlflag2;
    INTEGER   sfhdr.stctlhdr.ctlnumb1;
    INTEGER    sfhdr.stctlhdr.ctlnumb2;
    INTEGER   sfhdr.stctlhdr.ctlmsgk;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stctlhdr.ctlstat
Status type STCNTRL (0x02).
sfhdr.stctlhdr.ctlqual
Control qualifier ACKNEG2 (0x04).
sfhdr.stctlhdr.ctltype
Control type.
sfhdr.stctlhdr.ctlack
Reserved.
sfhdr.stctlhdr.ctlflag1
Reserved.
sfhdr.stctlhdr.ctlflag2
Reserved.
sfhdr.stctlhdr.ctlnumb1

Error code 1.

sfhdr.stctlhdr.ctlnumb2

Error code 2.

sfhdr.stctlhdr.ctlmsgk

Message key.

Remarks

- The **Status-Control(...) Negative-Acknowledge-2** message consists of a buffer header only; there is no buffer element.
- See the table in Status-Control Message for a summary of **Status-Control** control type codes.
- See Error and Sense Codes for a list of error codes.
- The message key field matches the message key in the corresponding **Status-Control** request.

Status-Control(...) ACKLUA

The **Status-Control(...) ACKLUA** message is for LUA applications only. It flows from the node to the application, and is used with a PLU connection.

```
struct Status-Control(...) ACKLUA {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stctlhdr.ctlstat;
    CHAR      sfhdr.stctlhdr.ctlqual;
    CHAR      sfhdr.stctlthdr.ctltype
    CHAR      sfhdr.stctlhdr.ctlack;
    CHAR      sfhdr.stctlhdr.ctlflag1;
    CHAR      sfhdr.stctlhdr.ctlflag2;
    INTEGER   sfhdr.stctlhdr.ctlnumb1;
    INTEGER   sfhdr.stctlhdr.ctlnumb2;
    INTEGER   sfhdr.stctlhdr.ctlmsgk;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stctlhdr.ctlstat
Status type STCNTRL (0x02).
sfhdr.stctlhdr.ctlqual
Control qualifier ACKLUA (0x05).
sfhdr.stctlhdr.ctltype
Control type.
sfhdr.stctlhdr.ctlack
Reserved.
sfhdr.stctlhdr.ctlflag1
Application flag 1.
sfhdr.stctlhdr.ctlflag2
Application flag 2.
sfhdr.stctlhdr.ctlnumb1

Code 1.

sfhdr.stctlhdr.ctlnumb2

Code 2.

sfhdr.stctlhdr.ctlmsgk

Message key (used for the SNA sequence number, see Remarks).

Remarks

- The **Status-Control(...) ACKLUA** message consists of a buffer header only; there is no buffer element.
- See the table in Status-Control Message for a summary of **Status-Control** control type codes.
- The application flags and the code 1 and code 2 fields are undefined and should not be used.
- The message key field gives the sequence number from the TH of the inbound data message to which this is an acknowledgment.

Status-Error

The **Status-Error** message is used to report "request reject" and "RH usage" error conditions in outbound SNA RUs to the application. It flows from the node to the application and is used with both SSCP and PLU connections.

See Status-Error Message for further information.

```
struct Status-Error {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.sterrhdr.errstat;
    CHAR      sfhdr.sterrhdr.errpad1;
    CHAR      sfhdr.sterrhdr.errpad2;
    CHAR      sfhdr.sterrhdr.errpad3;
    CHAR      sfhdr.sterrhdr.errcode1;
    CHAR      sfhdr.sterrhdr.errcode2;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.sterrhdr.errstat
Status type STERROR (0x03).
sfhdr.sterrhdr.errpad1
Reserved.
sfhdr.sterrhdr.errpad2
Reserved.
sfhdr.sterrhdr.errpad3
Reserved.
sfhdr.sterrhdr.errcode1
Error code 1.
sfhdr.sterrhdr.errcode2
Error code 2.

Remarks

- The **Status-Error** message consists of a buffer header only; there is no buffer element.
- The error codes are listed in Error and Sense Codes.

Status-Resource

The **Status-Resource** message is used to provide a simple flow control mechanism between the local node and the application to prevent the application from exhausting its resources. It flows from the application to the node, and is used with a PLU connection.

It is only used on the PLU connection where the application specifies in the PLU CICB that pacing requires application participation. See Pacing and Chunking for further details.

```
struct Status-Resource {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.streshdr.resstat;
    CHAR      sfhdr.streshdr.respad;
    CHAR      sfhdr.streshdr.rescred;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.streshdr.resstat
Status type STRESRCE (0x04).
sfhdr.streshdr.respad
Reserved.
sfhdr.streshdr.rescred
Application credit.

Remarks

- The **Status-Resource** message consists of a buffer header only; there is no buffer element.
- The **rescred** (application credit) field indicates that the application can receive a further "credit" RUs of the maximum RU size, or a further "credit" chunks if chunking is being used.

Status-RTM

The **Status-RTM** message provides the application with information on the Response Time Monitor (RTM) measurement parameters used by the host. This allows the application to match its local display of RTM statistics, if it provides such a display, with the statistics used by the host. It flows from the node to the application and is used with an SSCP connection.

See Response Time Monitor Data for further details.

```
struct Status-RTM {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.strtmhdr.rtmstat;
    CHAR      sfhdr.strtmhdr.strbndry;
    CHAR      sfhdr.strtmhdr.strcount;
    CHAR      sfhdr.strtmhdr.strtmdef;
    CHAR      sfhdr.strtmhdr.strtmact;
    CHAR      sfhdr.strtmhdr.strtmdsp;
};
```

Element

```
struct Status-RTM {
    PTRBFELT   hdreptr->elteptr;
    INTEGER    hdreptr->startd;
    INTEGER    hdreptr->endd;
    CHAR       hdreptr->trpad;
    CHAR[268]  hdreptr->dataru;
};
```

Members

nxtqptr
Pointer to next buffer header.

hdreptr
Pointer to buffer element.

numelts
Number of buffer elements.

msgtype
Message type STATFMI (0x21).

srcl
Source locality.

srcp
Source partner.

srci
Source index.

destl
Destination locality.

destp
Destination partner.

desti
Destination index.

sfhdr.strtmhdr.rtmstat
Status type STRTM (0x06).

sfhdr.strtmhdr.strbndry
RTM boundaries.

Possible values are:

0x00 - No RTM boundaries follow in element.
0x01 - RTM boundaries follow in element.

sfhdr.strtmhdr.strcount
RTM counters.

Possible values are:

0x00 - No RTM counters follow in element.
0x01 - RTM counters follow in element.

sfhdr.strtmhdr.strtmdef
RTM definition.

Possible values are:

0x00 - No change: use last received definition.
0x01 - Timers run until first data is written to screen.
0x02 - Timers run until keyboard is unlocked.
0x03 - Timers run until application can send (CD or EB received).

sfhdr.strtmhdr.strtmact
RTM measurement.

Possible values are:

0x00 - not active
0x01 - active

sfhdr.strtmhdr.strtmdsp
Local RTM display.

Possible values are:

0x00 - disabled
0x01 - enabled

Element

hdreptr–>elteptr
Pointer to buffer element (NIL).
hdreptr–>startd
Start of data in this element.
hdreptr–>endd
End of data in this element.
hdreptr–>trpad
Reserved.
hdreptr–>dataru
Data RU, as follows:

dataru[0–1] Number of boundaries in element

Possible values are:

0x0000 - no boundaries included

m - number of boundaries included

dataru[2–3] Number of counters in element

Possible values are:

0x0000 - no counters included

n - number of counters included

dataru[4–(2m+3)] m boundary values.

dataru[(2m+4)–(2m+2n+3)] n counter values.

dataru[(2m+2n+4) RTM total time.

Remarks

- A **Status-RTM** message is sent after the **Open(SSCP) OK Response** to give the initial RTM parameters. It is sent again when the RTM counters are reset (either on request from the host or when the local node sends unsolicited RTM data to the host), or when the host changes any of the RTM parameters.
- The message is sent only for applications that use LUs with type video display unit (VDU) or LUs in a VDU pool, since the RTM feature applies only to 3270 display sessions.
- All the values in the data RU are integer values.
- The RTM counter values in this message can be nonzero at startup, since RTM statistics are maintained for a specific LU and not for a specific application's use of that LU. If zero counter values are included, this indicates that the counters are to be reset.
- The RTM total time field is present only if the number of counters in the element is nonzero.

Status-Session

The **Status-Session** message provides the application with information on changes in the state of a session between the local node and the host. It flows from the node to the application and is used with both SSCP and PLU connections.

See Status-Session Message for further details.

```
struct Status-Session {
    PTRBFHDR  nxtqptr;
    PTRBFELT  hdreptr;
    CHAR      numelts;
    CHAR      msgtype;
    CHAR      srcl;
    CHAR      srcp;
    INTEGER   srci;
    CHAR      destl;
    CHAR      destp;
    INTEGER   desti;
    CHAR      sfhdr.stseshdr.sesstat;
    CHAR      sfhdr.stseshdr.sesspad;
    CHAR      sfhdr.steeshdr.sesscode;
    CHAR      sfhdr.stseshdr.sessqual;
};
```

Members

nxtqptr
Pointer to next buffer header.
hdreptr
Pointer to buffer element (NIL).
numelts
Number of buffer elements (0x00).
msgtype
Message type STATFMI (0x21).
srcl
Source locality.
srcp
Source partner.
srci
Source index.
destl
Destination locality.
destp
Destination partner.
desti
Destination index.
sfhdr.stseshdr.sesstat
Status type STSESSN (0x05).
sfhdr.stseshdr.sesspad
Reserved.
sfhdr.steeshdr.sesscode
Session code.
sfhdr.stseshdr.sessqual
Session code qualifier.

Remarks

- The **Status-Session** message consists of a buffer header only; there is no buffer element.
- The session codes are listed in Status-Session Codes.
- The SNA Server OS/2 and MS-DOS-based 3270 emulators display a session status identifier (–+z_*nnn*), where *nnn* is obtained by adding 484 to the value of **sessqual** from this message.

FMI Extension for the Windows Environment

This section describes the API extension to the Microsoft® Windows® 3270 Emulator Interface that converts link status and error codes to a printable string.

Following is a definition of the function, syntax, returns, and remarks for using the extension. Refer to FMI Status, Error, and Sense Codes for more information.

GetFmiReturnCode

The **GetFmiReturnCode** function converts link status and error codes to a printable string. This function provides a standard set of error strings for use by Function Management Interface (FMI) applications.

```
int WINAPI GetFmiReturnCode (
    UINT errcode1,
    UINT errcode2,
    UINT buffer_length,
    unsigned char FAR *buffer_addr
);
```

Parameters

errcode1
Supplied parameter; see Remarks.
errcode2
Supplied parameter; see Remarks.
buffer_length
Supplied parameter; specifies the length of the buffer pointed to by *buffer_addr*. The recommended length is 256.
buffer_addr
Supplied/returned parameter; specifies the address of the buffer that will hold the formatted, null-terminated string.

Return Values

0x20000001
The parameters are invalid; the function could not read the specified error codes or could not write to the specified buffer.
0x20000002
The specified buffer is too small.

Remarks

The *errcode1* and *errcode2* parameters are set according to the way that **GetFmiReturnCode** is used:

Codes to be translated	**Value for *errcode1***	**Value for *errcode2***
The *errcode1* and *errcode2* values specified in Error and Sense Codes which includes messages for: **Open(SSCP) Response**, **Open(PLU) Confirm**, **Status-Acknowledge(Nack-2)**, **Status-Control(...) Nack2**, **Status-Error**, and Appl-Data messages with SDI set	Unchanged from message	Unchanged from message
The status and qualifier codes returned from a Status-Session message	*status**256 + *qualifier*	0xFFFF
The return code from **WinLUAGetLastInitStatus**	The return code	0xFFFF

AFTP File Transfer Protocol

The Advanced Program-to-Program Communications (APPC) Application Suite provides application programming interfaces (APIs) to its APPC File Transfer Protocol (AFTP) functions. This guide provides the information you will need to write an application program to implement AFTP client functions. It is written for application programmers, and assumes that you are familiar with writing C-language applications. Although the APPC Application Suite API is based on APPC and the Common Programming Interface for Communications (CPI-C), you do not need to know these concepts to successfully write applications based on this API.

This section contains:

About the AFTP Guide

The Advanced Program-to-Program Communications (APPC) Application Suite provides application programming interfaces (APIs) to its APPC File Transfer Protocol (AFTP) functions. This guide provides the information you will need to write an application program to implement AFTP client functions. It is written for application programmers, and assumes that you are familiar with writing C-language applications. Although the APPC Application Suite API is based on APPC and the Common Programming Interface for Communications (CPI-C), you do not need to know these concepts to successfully write applications based on this API.

AFTP Programmer's Guide

The APPC File Transfer Protocol (AFTP) API is a set of C routines that provides file transfer capabilities for Advanced Program-to-Program Communications (APPC). This API makes file transfer programming easier by allowing you to access routines that interact with any AFTP server.

This section contains:

Defined Constants

The AFTP API has a number of calls that take buffers as parameters. Each buffer is used by one or more entry points and has a required minimum size that is defined by a constant in the AFTP API header file. When developing AFTP applications on Microsoft® Windows® 2000, Windows NT®, Windows 98, and Windows 95, this header file is called APPFFTP.H.

The following table provides information about these buffer parameters.

- The first column shows the entry points that take buffers as parameters.
- The second column shows the buffer used by each entry point.
- The third column shows the constant defining the minimum buffer size.
- The fourth column shows the minimum size.

Entry point	Buffer name	Constant name	Value
aftp_dir_open	path_buffer _length	AFTP_FILE_NAME _SIZE	512
aftp_dir_read	dir_entry_size	AFTP_FILE_NAME _SIZE	512
aftp_extract_destination	destination _size	AFTP_FQLU_NAME _SIZE	64
aftp_extract_mode _name	mode_name _size	AFTP_MODE_NAME _SIZE	8
aftp_extract_partner_lu _name	partner_LU _name_size	AFTP_FQLU_NAME _SIZE	64
aftp_extract_password	password_size	AFTP_PASSWORD _SIZE	10
aftp_extract_tp_name	tp_name_size	AFTP_TP_NAME _SIZE	64
aftp_extract_userid	userid_size	AFTP_USERID_SIZE	10
aftp_format_error	error_str_size	AFTP_MESSAGE _SIZE	2048
aftp_get_data_type _string	data_type_size	AFTP_DATA_TYPE _SIZE	64
aftp_get_date_mode _string	date_mode_size	AFTP_DATE_MODE _SIZE	64
aftp_get_record_format _string	record_format _size	AFTP_RECORD _FORMAT_SIZE	64
aftp_get_write_mode _string	write_mode_size	AFTP_WRITE_MODE _SIZE	64
aftp_local_dir_open	path_buffer _length	AFTP_FILE_NAME _SIZE	512
aftp_local_dir_read	dir_entry_size	AFTP_FILE_NAME _SIZE	512
aftp_query_local _system_info	system_info _size	AFTP_SYSTEM_INFO _SIZE	512
aftp_query_system_info	system_info _size	AFTP_SYSTEM_INFO _SIZE	512

Standard Types

Type definitions are available for many parameters to the AFTP API calls. For example, the AFTP type **AFTP_LENGTH_TYPE** is an alias for the C type **unsigned long**.

Use the AFTP types instead of the corresponding C types. Doing so protects you from changes to the parameters in future releases. If you use the AFTP types, you only need to recompile your code to use new API definitions. If you use the C types, you need to modify your program source code to reflect changes to new C types.

The AFTP API avoids complex structures and pointers to structures for type definitions. These complex structures might not be supported in all languages. The only exception is the string construct that is found in many languages.

The AFTP standard types are shown in the following table.

Type	Description
AFTP_HANDLE_TYPE	AFTP connection object identifier
AFTP_ALLOCATION_SIZE_TYPE	File allocation size
AFTP_BLOCK_SIZE_TYPE	File block size
AFTP_BOOLEAN_TYPE	Boolean type (FALSE=0, TRUE=1)
AFTP_DATA_TYPE_TYPE	File data types that can be transferred
AFTP_DATE_MODE_TYPE	Date mode used for transferred files
AFTP_DETAIL_LEVEL_TYPE	Amount of information to be output when AFTP generates error messages
AFTP_FILE_TYPE_TYPE	Kind of file (directory or file) listed by AFTP
AFTP_INFO_LEVEL_TYPE	Amount of information listed for a file by AFTP
AFTP_LENGTH_TYPE	Size of input buffers, and actual returned length of buffers in AFTP
AFTP_RETURN_CODE_TYPE	Return codes output by AFTP
AFTP_RECORD_FORMAT_TYPE	File record formats
AFTP_RECORD_LENGTH_TYPE	File record length
AFTP_SECURITY_TYPE	APPC security types
AFTP_TRACE_LEVEL_TYPE	Levels of tracing information output by AFTP
AFTP_VERSION_TYPE	AFTP program version numbers
AFTP_WRITE_MODE_TYPE	Different ways that AFTP can write to a file

Null-Terminated Strings

The AFTP API does not require input strings to be null-terminated and does not guarantee that output strings are null-terminated. If there is a null terminator, it is not included in the return size.

The C programmer should be aware of the fact that strings are handled differently in AFTP than they are in the C standard library. All API calls receiving strings as input require both the string itself and the length of the string. The strlen function can be used for this. The null terminator must not be counted as part of the string length. API calls that output strings require three string-related parameters:

- The string.
- The length of the string buffer that has been allocated by the calling program.
- The actual length of the string that is output. AFTP output strings are not null-terminated. For the C programmer to use them as standard C strings, a null character must be added to the end of the string.

AFTP_ENTRY

The AFTP API calls do not return a value. Rather, the *return_code* parameter is set to indicate the success or failure of the call. You should check the *return_code* parameter after each call and handle error values appropriately.

The C keyword void is not used for entry points in the AFTP API. Instead, AFTP_ENTRY has been defined. AFTP_ENTRY is defined differently depending on the operating system on which the AFTP client is created.

AFTP_PTR

The C pointer indicator (*****) is not used in the AFTP API. Instead, **AFTP_PTR** has been defined. **AFTP_PTR** is defined differently depending on the operating system on which the AFTP client is created.

AFTP Line Flows

The term "line flows" refers to the data formatting method used by the AFTP client and the AFTP server to communicate with each other. The AFTP API adheres to the AFTP line flow standards. All AFTP client applications send the same set of line flows over the network to the AFTP server. An application using the AFTP API does not need to be aware of the details of this line flow format.

AFTP File and Directory Concepts

The concept of a file and a directory can vary under the AFTP API depending on the supporting operating system. This section describes the AFTP file and directory concept as supported on the following operating environments:

- Microsoft Windows 2000
- Microsoft Windows NT
- Microsoft Windows 98
- Microsoft Windows 95

This section contains:

AFTP File Transfer Types

The AFTP API supports both text and binary file transfers. A binary file transfer treats a file as a stream of bytes. None of the characters within the file are interpreted. Executable programs and other nontext files are usually transferred as binary files.

In an ASCII file transfer, files are transferred using ASCII characters and the text file format is preserved. If either the source or destination is an EBCDIC computer (an IBM host system, for example), AFTP on the EBCDIC computer translates from ASCII to EBCDIC when it receives a file and from EBCDIC to ASCII when it sends a file.

> **Note** Text files on Windows 2000, Windows NT, Windows 98, Windows 95, OS/2, and Microsoft MS-DOS® can contain an end-of-file (EOF) character (0x1a) which is the last character in the file. When AFTP transfers a text file containing an EOF character, AFTP does not send the EOF character, so the file appears to be one character smaller when it is received by the server. If the file is transferred in binary mode, AFTP sends the EOF character.

AFTP and Special File Structures

If a file to be transferred has a special file structure based on record format and record length, AFTP may limit your ability to transfer that file.

If you want to send the file to a system that supports the record format and record length of the file, use the aftp_set_record_format and aftp_set_record_length functions first to ensure the file is correctly transferred.

If you send the file to a system that does not maintain the record format and length, you may lose critical information and be unable to restore it when the file is copied back to the client.

> **Note** You might be able to use an archiving tool to convert the special file into a format that can be transferred to other platforms. When the file is to be used again on the original platform, use the archiving tool to restore the file to its original format.

Working with AFTP Directories

In the AFTP API, directories represent the structures used to divide a file system into multiple portions. An AFTP directory can contain one or more files. Directories are organized in a hierarchical structure. The topmost node in the directory structure is called the root.

The AFTP API allows you to designate a directory as the current directory. The primary advantage of setting a current directory is that when you are working with files in this directory, it is not necessary to include the full directory specification in AFTP API functions.

Directories on Windows 2000, Windows NT, Windows 98, and Windows 95

The native Windows 2000, Windows NT, Windows 98, and Windows 95 operating systems use a hierarchical directory structure based on the drive letter. The AFTP API builds on this structure by making the drive the first segment of the directory path. For an AFTP application operating on Windows 2000, Windows NT, Windows 98, or Windows 95, the AFTP root directory represents all of the drives available on the computer. In the native Windows 2000, Windows NT, Windows 98, or Windows 95 operating system, a current directory setting is maintained for each drive. This difference of interpretation can have some surprising consequences.

For example, the following series of commands issued on Windows 2000, Windows NT, Windows 98, or Windows 95 will delete the files in the C:\WORK\AFTP directory:

```
[C:\]  cd \work\aftp
[C:\WORK\AFTP\]  cd e:\new\data\
[C:\WORK\AFTP\]  erase C:*.*
```

AFTP, however, uses a virtual root that encompasses all of the drives, and the Windows 2000, Windows NT, Windows 98, or Windows 95 drive letter is treated as the first segment of a path. Only one current directory setting is maintained by the AFTP API. If a path is not specified to an AFTP API function, the root directory of that drive is used.

The equivalent set of functions issued using the AFTP API will erase all of the files on drive C, not only the files in the C:\WORK\AFTP directory:

```
aftp> cd c:\work\aftp
aftp> cd e:\new\data
aftp> delete c:*.*
```

If it becomes necessary to manipulate files on a drive other than the drive set as the current directory, always use fully qualified file names to ensure that the correct directory and files are used.

Compiling the AFTP Application

This section describes procedures for developing and compiling an application for the AFTP API on Microsoft Windows 2000, Windows NT, Windows 98, Windows 95, IBM MVS, and IBM VM.

To develop an application that uses the AFTP API on Windows 2000, Windows NT, Windows 98, or Windows 95

1. Include the APPFFTP.H header file in your source modules. For consistency with the file naming conventions on other platforms, you can rename this header file to AFTPAPI.H and use this file name for your AFTP include file.
2. Define CM_WINNT when you compile your source code.

To develop an application that uses the AFTP API on MVS

> **Note** This process assumes that all AFTP program files have been successfully installed with the JCL provided for installing AFTP.

1. Include the APPFFTP.H header file in your source modules.
2. Define CM_MVS when you compile your source code.
3. Edit the APPFAPIJ JCL file and make the changes indicated in the prolog comments at the top of the file.
4. Submit the APPFAPIJ JCL.

To access the AFTP API calls from your application using VM

1. Include the APPFFTP.H header file in your source modules.
2. Define CM_VM when you compile your source code.
3. Add APPFAPIL TXTLIB to your GLOBAL TXTLIB statement. All textdecks are packaged into this textlib.

Linking the AFTP Application

This section describes the procedure for linking an application for the AFTP API on Microsoft Windows 2000, Windows NT, Windows 98, or Windows 95.

To link an application that uses the AFTP API on Windows 2000, Windows NT, Windows 98, or Windows 95

1. The AFTP application must be statically linked with the AFTPAPI.LIB import library supplied as part of the Host Integration Server 2000 SDK or the earlier SNA Server 4.0 SDK.
2. Include the appropriate AFTPAPI.DLL with your application when it is installed on the target machine if this DLL is not already installed.

Note that Host Integration Server 2000 does not support the DEC Alpha and the Host Integration Server SDK does not include the DEC Alpha version of the AFTPAPI.LIB and AFTPAPI.DLL files. The DEC Alpha versions of these files are included as part of the earlier SNA Server 4.0 SDK.

Overview of API Calls

This section provides a brief description of each AFTP API call.

This section contains:

Creating or Destroying an AFTP Connection Object

The connection object represents an object-oriented approach to AFTP. An AFTP connection object represents a connection (not necessarily active) to a partner computer. Many of the AFTP API calls require an AFTP connection object as input. When the program has finished using AFTP API calls, it should destroy the connection object.

aftp_create
: Creates an AFTP connection object and assigns a unique identifier to it. The connection object is accessed by its connection identifier. The connection object is never automatically destroyed. This allows you to connect to an AFTP server once or reconnect numerous times using the same AFTP connection object.

aftp_destroy
: Destroys the AFTP connection object and recovers all resources associated with it. When an AFTP connection object has been destroyed, that object must not be used again. If you need a connection object again, create another one.

Establishing a Connection to the AFTP Server Computer

For the AFTP client to communicate with an AFTP server, certain communications parameters must be set. Most of the communications parameters have default values. The destination does not have a default value and must be set explicitly. When the parameters are set as desired, the connection to the server can take place.

Use these API calls to manage your connection to the AFTP server.

aftp_close
: Closes a connection to the AFTP server when processing is complete.

aftp_connect
: Establishes the connection to the AFTP server for file transfer.

aftp_set_destination
: Identifies the server computer name to which the AFTP connection will be established. This server computer will run the AFTP server program.

aftp_set_mode_name
: Sets the mode name to be used for this connection. The default mode name is #BATCH.

aftp_set_password
: Sets the password used for APPC security type PROGRAM. Using this call automatically sets the security type to PROGRAM.

aftp_set_security_type
: Sets the APPC security used for the AFTP connection to the AFTP server.

aftp_set_tp_name
: Sets the transaction program (TP) name to be used for this connection. The default TP name is AFTPD.

aftp_set_userid
: Sets the user identifier used for APPC security type PROGRAM. Using this call automatically sets the security type to PROGRAM.

Querying Connection Characteristics

Use these API calls to query the characteristics of the connection to the AFTP server.

aftp_extract_destination
: Extracts the identity of the server computer on which the AFTP server runs.

aftp_extract_mode_name
: Extracts the mode name used for this connection.

aftp_extract_partner_lu_name
: Extracts the fully qualified logical unit name of the AFTP server computer.

aftp_extract_password
: Extracts the password used for this connection.

aftp_extract_security_type
: Extracts the security type used for this connection.

aftp_extract_tp_name
: Extracts the transaction program name used for this connection.

aftp_extract_userid
: Extracts the user identifier used for this connection.

Transferring Files

The primary purpose of the file transfer protocol is to exchange files between the AFTP client and AFTP server programs. Through the API, the AFTP client program can send a file to the AFTP server and receive a file from the AFTP server.

Use these API calls to transfer files between the client and server.

aftp_query_bytes_transferred
: Outputs the number of bytes transferred by either the **aftp_send_file** or **aftp_receive_file** call.

aftp_receive_file
: Receives a single file from the AFTP server.

aftp_send_file
: Sends a single file to the AFTP server.

Specifying File Transfer Characteristics

AFTP supports a variety of file transfer attributes. Both text and binary files can be transferred. The programmer can set the data structure of the files for mainframe applications. This allows several variations of record-based files to be transferred.

Use these API calls to specify file transfer characteristics.

aftp_set_allocation_size
: Sets the amount of space allocated for the file that is being written.

aftp_set_block_size
: Sets the size of a data block for the file that is being written.

aftp_set_data_type
: Sets the way in which the transmitted data is interpreted.

aftp_set_date_mode
: Sets how the date will be represented when the file is written (either received or sent). The new file can use the current date/time stamp or the date/time stamp of the original file.

aftp_set_record_format
: Sets the record format of the transmitted file.

aftp_set_record_length
: Sets the length of the transmitted file record.

aftp_set_write_mode
: Sets the type of write operation that will occur when the transmitted file is written (either received or sent).

Querying File Transfer Characteristics

Use these API calls to query the file transfer attributes.

aftp_extract_allocation_size
: Extracts the amount of space allocated for the file that is being transmitted.

aftp_extract_block_size
: Extracts the size of a data block for the file that is being transmitted.

aftp_extract_data_type
: Extracts the way in which the transmitted data is interpreted.

aftp_extract_date_mode
: Extracts how the date will be represented when the file is written.

aftp_extract_record_format
: Extracts the record format of the transmitted file.

aftp_extract_record_length
: Extracts the length of the transmitted file record.

aftp_extract_write_mode
: Extracts the type of write operation that will occur when the transmitted file is written.

Listing Files on the AFTP Server Computer

File list facilities can be used to support wildcard transfers from the AFTP server. Wildcard processing is not allowed by the aftp_send_file and aftp_receive_file functions to make these functions as portable as possible. Obtaining a complete directory listing requires three calls: open, read, and close. The read function can be called multiple times to support wildcard processing.

aftp_dir_close
: Closes an active directory listing on the AFTP server.

aftp_dir_open
: Begins a directory listing operation on the AFTP server. The directory open call sets up the search specifications:

  - File specification that is to be matched.
  - Whether directories, files, or both should be included in the search.
  - The type of information desired (file names only or file names with attributes).

aftp_dir_read
: Gets the next file from the directory listing on the AFTP server. A text string describing the file is returned. The format of the information returned depends on the parameters specified on the **aftp_dir_open** call.

Listing Files on the AFTP Client Computer

File list facilities can be used to support wildcard transfers to the AFTP server. Wildcard processing is not allowed by the aftp_send_file and aftp_receive_file functions to make these functions as portable as possible. Obtaining a complete directory listing requires three calls: open, read, and close. The read function can be called multiple times to support wildcard processing.

aftp_local_dir_close
: Closes an active directory listing on the AFTP client.

aftp_local_dir_open
: Begins a directory listing operation on the AFTP client. The directory open call sets up the search specifications:

  - File specification to be matched.
  - Whether directories, files, or both should be included in the search.
  - The type of information desired (file names only or file names with attributes).

aftp_local_dir_read
: Gets the next file from the directory listing on the AFTP client. A text string describing the file is returned. The format of the information returned depends upon the parameters specified on the **aftp_local_dir_open** call.

Performing Directory Manipulation

Server Directories:

AFTP provides methods of traversing and modifying the directory structure on the AFTP server computer. It is possible to build recursive copy routines for entire directory trees using these calls. AFTP also maintains the current directory on the AFTP server. This enables the user to specify a file name without specifying the entire directory path to that file on the AFTP server.

Use these API calls to manage directories on the server.

aftp_change_dir
: Changes the current working directory on the AFTP server.

aftp_create_dir
: Makes a new directory on the AFTP server.

aftp_query_current_dir
: Outputs the current working directory on the AFTP server.

aftp_remove_dir
: Removes an existing directory on the AFTP server.

Client Directories:

AFTP provides methods to query and traverse the directory structure on the AFTP client computer. AFTP maintains the current directory on the AFTP client. This enables the user to specify a file name without specifying the entire directory path to that file on the AFTP client.

Use these API calls to manage directories on the client.

aftp_local_change_dir
: Changes the current working directory on the AFTP client.

aftp_local_query_current_dir
: Outputs the current working directory on the AFTP client.

Performing File Manipulation

The following two calls provide additional file functions that allow modifications to files on the AFTP server without using the aftp_send_file call. It is possible to rename a file on the AFTP server computer as long as the rename does not cross device boundaries. It is also possible to delete files on the AFTP server computer.

aftp_delete
: Deletes a file on the AFTP server.

aftp_rename
: Renames a file on the AFTP server.

Querying System Information

The query system calls can be used to obtain more information about the AFTP client and AFTP server computers.

aftp_query_local_system_info
: Outputs a string describing the AFTP client operating system.

aftp_query_local_version
: Outputs the major and minor AFTP client version numbers.

aftp_query_system_info
: Outputs a string describing the AFTP server operating system.

Generating Message Strings

AFTP allows the caller to use consistent strings for AFTP transfer characteristics. AFTP will output the string to use when queried. It is also possible to output standard text messages for AFTP errors. The other API calls return an AFTP return code that can be queried to determine if an error message should be output.

Use these API calls to get text strings to use in messages issued by your application.

aftp_format_error
: Generates text output for the current AFTP error. This should be used to output error information to the user when an AFTP call returns an error code.

aftp_get_data_type_string
: Outputs the string corresponding to an input data type value.

aftp_get_date_mode_string
: Outputs the string corresponding to an input date mode value.

aftp_get_record_format_string
: Outputs the string corresponding to an input record format value.

aftp_get_write_mode_string
: Outputs the string corresponding to an input write mode value.

Controlling Trace Information

Use these API calls to control tracing of AFTP activity.

aftp_extract_trace_level
: Extracts the current trace level.

aftp_set_trace_filename
: Sets the name of the file to be used for trace output.

aftp_set_trace_level
: Sets the amount of trace data to be captured.

Loading the Initialization File

AFTP provides a method of loading the AFTP initialization file that contains user permission and file mapping information.

aftp_load_ini_file
: Loads the AFTP initialization file into memory.

AFTP Reference

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides information about the APPC File Transfer Protocol (AFTP) calls, return codes, and entry point mappings.

This section contains:

AFTP API Call Reference

This section provides an alphabetic reference for all of the API calls for the APPC File Transfer Protocol (AFTP). Code fragments are provided for each call to illustrate its use.

aftp_change_dir

The **aftp_change_dir** call changes the current working directory on the AFTP server. A connection to the AFTP server must be established before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_change_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The new current working directory name. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The directory specified can be either an absolute or a relative path name.

length
The length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    /* The value used will vary based on platform:
     *   VM common naming:       directory = "/d"
     *   VM native naming:       directory = "/d"
     *   MVS PDS common naming:  directory = "/user.clist/"
     *   MVS PDS native naming:  directory = "'user.clist'"
     *   MVS data set prefix common:  directory = "/user.qual.a."
     *   MVS data set prefix native:  directory =
         "'user.qual.a.'"
     *   NT* common naming:    directory = "/c:/nt"
     *   NT native naming:     directory = "c:\\nt"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/"; /*
    MVS */

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

     /*
      * Specify the new current working directory name
      * using the COMMON name format.
      */

     aftp_change_dir(
         connection_id,
         directory,
         (AFTP_LENGTH_TYPE)strlen(directory),
         &aftp_rc);

     if (aftp_rc != AFTP_RC_OK) {
         fprintf(stderr, "Error changing AFTP directory.\n");
```

```
        }
    }
```

Line Flows

The directory name is sent to the AFTP server and the call waits for a response indicating the success or failure of the change directory operation.

aftp_close

The **aftp_close** call closes an active connection. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_close(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

     aftp_close(connection_id, &aftp_rc);
     if (aftp_rc != AFTP_RC_OK) {
         fprintf(
             stderr,
             "Error on aftp_close(): %s\n",
             aftp_rc);
}
```

Line Flows

The close operation causes a DEALLOCATE verb to be issued with an AP_FLUSH parameter, forcing any buffer contents to flow to the server before deallocating the conversation.

aftp_connect

The **aftp_connect** call establishes a connection to the AFTP server. You must identify the destination of the AFTP server with the aftp_set_destination call before issuing this call.

```
AFTP_ENTRY aftp_connect(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     * a connection_id, use:  aftp_create()
     * a destination, use:    aftp_set_destination()
     */

    /*
     * Establish a connection to the server
     */

    aftp_connect(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(
            stderr,
            "Error on aftp_connect(): %s\n",
            aftp_rc);
    }
}
```

Line Flows

The **aftp_connect** call causes an ALLOCATE verb to be issued to the server. When a conversation is established, an exchange of version numbers and capabilities occurs between the client and the server. Therefore, this call does not return until either AFTP verifies that the server program is running correctly on the remote system or an error occurs.

aftp_create

The **aftp_create** call creates an AFTP connection object that can be used to connect to an AFTP server.

```
AFTP_ENTRY aftp_create(
    OUT AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
Handle of the AFTP connection object that was created by this call. All subsequent AFTP calls must use a previously-created AFTP connection object.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_HANDLE_TYPE      connection_id;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Create the connection object that we will use for AFTP.
     */

    aftp_create(connection_id, &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error creating an AFTP object.\n");
    }
}
```

Line Flows

None.

aftp_create_dir

The **aftp_create_dir** call creates a new directory on the AFTP server. A connection to the AFTP server must be established before using this call.

Platform differences are as follows:

- On VM, this call is not supported. If issued, the call fails with return code AFTP_RC_FAIL_NO_RETRY.
- On MVS, partitioned data sets act as the directory structure. This call creates a partitioned data set with the name specified.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_create_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The name of the directory to be created. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The directory specified can be either an absolute or relative path name.

length
The length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM  not supported
     *   MVS PDS common naming:  directory="/user.clist/"
     *   MVS PDS native naming:  directory="'user.clist'"
     *   NT native naming:     directory="d:\\newdir"
     *   NT common naming:     directory="/d:/newdir"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/";


    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    aftp_create_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)strlen(directory),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error creating directory.\n");
    }
}
```

Line Flows

The directory name is sent to the AFTP server and the call waits for a response indicating the success or failure of the create directory operation.

aftp_delete

The **aftp_delete** call deletes a single file on the AFTP server. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_delete(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR filename,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

filename
The name of the file to be removed. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can contain either an absolute or relative path name.

length
The length of the *filename* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filespec="/a/foo*"
     *   VM native naming:       filespec="foo*.*.a"
     *   MVS PDS common naming:  filespec="/user.clist/foo*"
     *   MVS PDS native naming:  filespec="'user.clist(foo*)'"
     *   MVS sequential common:  filespec="/user.qual*.a*.**"
     *   MVS sequential native:  filespec="'user.qual*.a*.**'"
     */
    static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    /*
     * Delete a file
     */

    aftp_delete(
        connection_id,
        filespec,
        (AFTP_LENGTH_TYPE)strlen(filespec),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error deleting AFTP file.\n");
    }

}
```

Line Flows

The file name is sent to the AFTP server and the call waits for a response indicating the success or failure of the delete file operation.

aftp_destroy

The **aftp_destroy** call destroys an AFTP connection object. When an AFTP connection object is destroyed, it cannot be used again.

You should issue the aftp_close call to end the connection before you issue this call.

```
AFTP_ENTRY aftp_destroy(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object to be destroyed. This object was originally created with aftp_create.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE aftp_rc;
    AFTP_HANDLE_TYPE      connection_id;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * If you have opened a connection with aftp_connect()
     * you must also issue an aftp_close()
     */

     aftp_destroy(connection_id, &aftp_rc);

}
```

Line Flows

None.

aftp_dir_close

The **aftp_dir_close** call cancels a directory listing that is in progress on the AFTP server or ends a directory listing on the AFTP server after a nonzero *no_more_entries* has been returned from an aftp_dir_read call. A connection to the AFTP server must be established before using this call. A directory listing on the AFTP server must be started by calling aftp_dir_open before issuing this call.

```
AFTP_ENTRY aftp_dir_close(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR filespec,
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

See aftp_dir_read for a complete example showing the related calls **aftp_dir_open**, **aftp_dir_read**, and **aftp_dir_close**.

Line Flows

None.

aftp_dir_open

The **aftp_dir_open** call begins a directory listing and specifies the file search parameters on the AFTP server. The aftp_dir_read call is used to read individual directory entries. The aftp_dir_close call is used to end the directory listing. A connection to the AFTP server must be established before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_dir_open(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR filespec,
    IN AFTP_LENGTH_TYPE length,
    IN AFTP_FILE_TYPE_TYPE file_type,
    IN AFTP_INFO_LEVEL_TYPE info_level,
    OUT unsigned char AFTP_PTR path,
    IN AFTP_LENGTH_TYPE path_buffer_length,
    OUT AFTP_LENGTH_TYPE AFTP_PTR path_returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

filespec
The search string that the server uses to generate the directory listing. The files in the listing must match the search string. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can be either an absolute or relative path name and can contain wildcard characters.

length
The length of the *filespec* parameter in bytes.

file_type
The type of information (directory names or file names) to be returned.

AFTP_FILE Only file entries.

AFTP_DIRECTORY Only directory entries.

AFTP_ALL_FILES Both file and directory entries.

info_level
The level and format of information to be returned about each file or directory entry.

AFTP_NATIVE_NAMES Native names without attributes.

AFTP_NATIVE_ATTRIBUTES Native names and native file attributes.

path
The fully qualified directory name in which of all of the directory entries exist. The actual directory entries will be returned when the aftp_dir_read call is used. The path can be used along with the returned directory entry file name to create a fully qualified path name to use on another AFTP file call.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

path_buffer_length
The size in bytes of the buffer pointed to by the *path* parameter.

path_returned_length
The number of bytes returned in the *path* parameter.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

See aftp_dir_read for a complete example showing the related calls **aftp_dir_open**, **aftp_dir_read**, and **aftp_dir_close**.

Line Flows

Sends a request for a directory listing to the AFTP server and waits for a response that includes the fully specified path of the directory listing or an error indicator. If the path of the directory listing is received, the AFTP server sends all of the directory entries as well. When the list is complete, a special end-of-list indicator is sent to the AFTP client.

aftp_dir_read

The **aftp_dir_read** call gets an individual directory entry based on the search parameters specified in the aftp_dir_open call. A connection to the AFTP server must be established before using this call. The **aftp_dir_open** call must be issued before listing the directory entries.

```
AFTP_ENTRY aftp_dir_read(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR dir_entry,
    IN AFTP_LENGTH_TYPE dir_entry_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_BOOLEAN_TYPE AFTP_PTR no_more_entries,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

dir_entry
Pointer to a buffer into which the procedure will write the directory entry.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

dir_entry_size
The size in bytes of the *dir_entry* buffer.

returned_length
The number of bytes returned in the *dir_entry* buffer.

no_more_entries
Whether or not an entry was returned on this call.

A value of zero indicates that there are more directory entries and that an entry was returned on this call.

A nonzero value indicates that there are no more directory entries and no entry was returned on this call. The *returned_length* parameter is set to zero. Subsequent calls to aftp_dir_read will also result in *no_more_entries* being nonzero. To end the directory listing, your next call should be aftp_dir_close.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

This example shows how to use the **aftp_dir_open**, **aftp_dir_read**, and **aftp_dir_close** calls together.

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 dir_entry[AFTP_FILE_NAME_SIZE +1];
    AFTP_LENGTH_TYPE              dir_entry_length;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filespec="/a/foo*"
     *   VM native naming:       filespec="foo*.*.a"
     *   MVS PDS common naming:  filespec="/user.clist/foo*"
     *   MVS PDS native naming:  filespec="'user.clist(foo*)'"
     *   MVS sequential common:  filespec="/user.qual*.a*.**"
     *   MVS sequential native:  filespec="'user.qual*.a*.**'"
     */
    static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

    unsigned char                 path[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE              path_length;
    AFTP_BOOLEAN_TYPE             no_more_entries;

    /*
```

```
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    /*
     * Open a new directory listing on the AFTP server. Both files and
     * directory names will be listed along with their attributes.
     */

    aftp_dir_open(
        connection_id,
        filespec,
        (AFTP_LENGTH_TYPE)strlen(filespec),
        AFTP_DIRECTORY | AFTP_FILE,
        AFTP_NATIVE_ATTRIBUTES,
        path,
        (AFTP_LENGTH_TYPE)sizeof(path)-1,
        &path_length,
        &aftp_rc);

    if (aftp_rc == AFTP_RC_OK) {
        path[path_length] = '\0';

        printf("Directory listing of %s.", path);

        do {
            /*
             * Read one directory entry from the AFTP server
             */

            aftp_dir_read(
                connection_id,
                dir_entry,
                (AFTP_LENGTH_TYPE)sizeof(dir_entry)-1,
                &dir_entry_length,
                &no_more_entries,
                &aftp_rc);

            if (aftp_rc == AFTP_RC_OK && no_more_entries == 0) {
                dir_entry[dir_entry_length] = '\0';
                printf("File: %s\n", dir_entry);
            }
            /*
             * Loop until we either run out of directory
             * entries or an error occurs.
             */

        } while (aftp_rc == AFTP_RC_OK && no_more_entries == 0);

        /*
         * Terminate the directory listing by executing
         * a close.
         */

        aftp_dir_close(connection_id, &aftp_rc);
        if (aftp_rc != AFTP_RC_OK) {
            fprintf(
                stderr,
                "Error closing AFTP directory.\n");
        }
    }
    else {
        fprintf(stderr, "Error opening AFTP directory.\n");
    }
}
```

Line Flows

None.

aftp_extract_allocation_size

The **aftp_extract_allocation_size** call extracts the AFTP file allocation size. If the aftp_set_allocation_size call has not been invoked, the AFTP default allocation size value is returned.

```
AFTP_ENTRY aftp_extract_allocation_size(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_ALLOCATION_SIZE_TYPE AFTP_PTR allocation_size,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
allocation_size
The allocation size in bytes that was set for the AFTP file transfer operation.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE           connection_id;
    AFTP_RETURN_CODE_TYPE      aftp_rc;
    AFTP_ALLOCATION_SIZE_TYPE  allocation_size;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    aftp_extract_allocation_size(
        connection_id,
        &allocation_size,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP allocation size.\n");
    }
}
```

Line Flows

None.

aftp_extract_block_size

The **aftp_extract_block_size** call extracts the file block size. If the aftp_set_block_size call has not been invoked, the AFTP default block size value is returned.

```
AFTP_ENTRY aftp_extract_block_size(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_BLOCK_SIZE_TYPE AFTP_PTR block_size,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
block_size
The AFTP file block size in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    AFTP_BLOCK_SIZE_TYPE     block_size;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    aftp_extract_block_size(
        connection_id,
        &block_size,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP block size.\n");
    }

}
```

Line Flows

None.

aftp_extract_data_type

The **aftp_extract_data_type** call extracts the data type for file transfers. If the aftp_set_data_type call has not been invoked, the AFTP default data type value is returned.

```
AFTP_ENTRY aftp_extract_data_type(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_DATA_TYPE_TYPE AFTP_PTR data_type,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

data_type
The data type to be used for data transfers.

AFTP_ASCII Transfer files as text files in ASCII.

AFTP_BINARY Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE Use the data transfer type set in the .INI file. If no type is set in the .INI file, use AFTP_ASCII.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE        connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;
    AFTP_DATA_TYPE_TYPE     data_type;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    aftp_extract_data_type(
        connection_id,
        &data_type,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP data type.\n");
    }

}
```

Line Flows

None.

aftp_extract_date_mode

The **aftp_extract_date_mode** call extracts the way file dates are handled for data transfer. If the aftp_set_date_mode call has not been invoked, the AFTP default date mode value is returned.

```
AFTP_ENTRY aftp_extract_date_mode(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_DATE_MODE_TYPE AFTP_PTR date_mode,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

date_mode
The type of date given to the new file after transfer.

AFTP_NEWDATE Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE Assign the time/date stamp of the source file.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE           connection_id;
    AFTP_RETURN_CODE_TYPE      aftp_rc;
    AFTP_DATE_MODE_TYPE        date_mode;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    aftp_extract_date_mode(
        connection_id,
        &date_mode,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP date mode.\n");
    }

}
```

Line Flows

None.

aftp_extract_destination

The **aftp_extract_destination** call extracts the destination of the AFTP server. If the aftp_set_destination call has not been invoked, the AFTP default destination value is returned.

```
AFTP_ENTRY aftp_extract_destination(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR destination,
    IN AFTP_LENGTH_TYPE destination_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

destination
Buffer into which the name of the AFTP server is written. This parameter can be either a symbolic destination name or a partner LU name.

See the *APPC Application Suite User's Guide* for information about specifying destinations in the APPC Application Suite.

destination_size
The size of the buffer in which the destination will be stored.

Use the AFTP_FQLU_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

returned_length
The actual length of the *destination* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 destname[AFTP_FQLU_NAME_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the destination name for AFTP.
     */

    aftp_extract_destination(
        connection_id,
        destname,
        (AFTP_LENGTH_TYPE)sizeof(destname)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP destination.\n");
    }
}
```

Line Flows

None.

aftp_extract_mode_name

The **aftp_extract_mode_name** call extracts the mode name specified for the connection to the AFTP server. If the aftp_set_mode_name call has not been invoked, the AFTP default mode name value is returned.

```
AFTP_ENTRY aftp_extract_mode_name(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR mode_name,
    IN AFTP_LENGTH_TYPE mode_name_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

mode_name
The buffer in which the mode name is to be stored.

Use the AFTP_MODE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

mode_name_size
The size of buffer in which the mode name will be stored.

returned_length
The actual length of the *mode_name* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE                connection_id;
    AFTP_RETURN_CODE_TYPE           aftp_rc;
    unsigned char AFTP_PTR          mode_name[AFTP_MODE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE                returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the APPC mode name for AFTP.
     */

    aftp_extract_mode_name(
        connection_id,
        mode_name,
        (AFTP_LENGTH_TYPE)sizeof(mode_name)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP mode name.\n");
    }
}
```

Line Flows

None.

aftp_extract_partner_lu_name

The **aftp_extract_partner_lu_name** call extracts the fully qualified LU name of the server. A connection to the AFTP server must occur before this call can be invoked.

```
AFTP_ENTRY aftp_extract_partner_lu_name(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR partner_lu_name,
    IN AFTP_LENGTH_TYPE partner_lu_name_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

partner_lu_name
Buffer to which the fully qualified LU name is written.

Use the AFTP_FQLU_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

partner_lu_name_size
The size of the buffer to which the partner LU name will be written.

returned_length
The actual length of the *partner_lu_name* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            partner[AFTP_FQLU_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    /*
     * Extract the partner LU name for AFTP.
     */

    aftp_extract_partner_lu_name(
        connection_id,
        partner,
        (AFTP_LENGTH_TYPE)sizeof(partner)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP destination.\n");
    }
}
```

Line Flows

None.

aftp_extract_password

The **aftp_extract_password** call extracts the password specified for the connection to the AFTP server. If the aftp_set_password call has not been invoked, the AFTP default password value is returned.

```
AFTP_ENTRY aftp_extract_password(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR password,
    IN AFTP_LENGTH_TYPE password_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

password
The buffer in which the password used on the connection is written.

Use the AFTP_PASSWORD_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

password_size
The size of the buffer in which the password is to be written.

returned_length
The actual length of the *password* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 password[AFTP_PASSWORD_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the password for AFTP.
     */

    aftp_extract_password(
        connection_id,
        password,
        (AFTP_LENGTH_TYPE)sizeof(password)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP password.\n");
    }

}
```

Line Flows

None.

aftp_extract_record_format

The **aftp_extract_record_format** call extracts the record format for the data transfer. If the aftp_set_record_format call has not been invoked, the AFTP default record format value is returned.

```
AFTP_ENTRY aftp_extract_record_format(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RECORD_FORMAT_TYPE AFTP_PTR record_format,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

record_format
The record format used for file transfer.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FBA

Fixed length record, blocked, ASA print-control characters.

AFTP_FBM

Fixed length record, blocked, machine print-control codes.

AFTP_FBS

Fixed length record, blocked, standard.

AFTP_FBSA

Fixed length record, blocked, standard, ASA print-control characters.

AFTP_FBSM

Fixed length record, blocked, standard, machine print-control codes.

AFTP_FM

Fixed length record, unblocked, machine print-control codes.

AFTP_U

Undefined length record.

AFTP_UA

Undefined length record, ASA print-control characters.

AFTP_UM

Undefined length record, machine print-control codes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE          connection_id;
    AFTP_RETURN_CODE_TYPE     aftp_rc;
    AFTP_RECORD_FORMAT_TYPE   record_format;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the file record format for AFTP.
     */

    aftp_extract_record_format(
        connection_id,
        &record_format,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP record format.\n");
    }
}
```

Line Flows

None.

aftp_extract_record_length

The **aftp_extract_record_length** call extracts the record length for fixed length records, or the maximum possible record length for variable length records used for data transfer. If the aftp_set_record_length call has not been invoked, the AFTP default record length value is returned.

```
AFTP_ENTRY aftp_extract_record_length(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RECORD_LENGTH_TYPE AFTP_PTR record_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
record_length
The record length for the data transfer specified in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE            connection_id;
    AFTP_RETURN_CODE_TYPE       aftp_rc;
    AFTP_RECORD_LENGTH_TYPE     record_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the file record length for AFTP.
     */

    aftp_extract_record_length(
        connection_id,
        &record_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
       fprintf(stderr, "Error extracting AFTP record length.\n");
    }
}
```

Line Flows

None.

aftp_extract_security_type

The **aftp_extract_security_type** call extracts the type of APPC conversation security used. If the aftp_set_security_type call has not been invoked, the AFTP default security type value is returned.

```
AFTP_ENTRY aftp_extract_security_type(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_SECURITY_TYPE AFTP_PTR security_type,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

security_type
The security to be used when connecting to the AFTP server.

AFTP_SECURITY_NONE

No APPC conversation security is used.

AFTP_SECURITY_SAME

The local security information determined at logon time will be transferred to the AFTP server.

AFTP_SECURITY_PROGRAM

A user identifier and password will be sent to be verified by the AFTP server. You must use the aftp_set_userid and aftp_set_password calls with this security type, or the connection attempt will fail.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    AFTP_SECURITY TYPE            sec_type;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the APPC security type for AFTP.
     */

    aftp_extract_security_type(
        connection_id,
        &sec_type,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr,
        "Error extracting AFTP security type.\n");
    }
}
```

Line Flows

None.

aftp_extract_tp_name

The **aftp_extract_tp_name** call extracts the transaction program (TP) name of the AFTP server. If the aftp_set_tp_name call has not been invoked, the AFTP default TP name value is returned.

```
AFTP_ENTRY aftp_extract_tp_name(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR tp_name,
    IN AFTP_LENGTH_TYPE tp_name_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

tp_name
The buffer into which the TP name of the AFTP server will be written.

Use the AFTP_TP_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

tp_name_size
The size of the buffer to which the TP name will be written.

returned_length
The actual length of the *tp_name* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE                connection_id;
    AFTP_RETURN_CODE_TYPE           aftp_rc;
    unsigned char                   tp_name[AFTP_TP_NAME_SIZE+1];
    AFTP_LENGTH_TYPE                returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the TP name for AFTP.
     */

    aftp_extract_tp_name(
        connection_id,
        tp_name
        (AFTP_LENGTH_TYPE)sizeof(tp_name)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP TP name.\n");
    }
}
```

Line Flows

None.

aftp_extract_trace_level

The **aftp_extract_trace_level** call extracts the current trace level setting. If the aftp_set_trace_level call has not been invoked, the AFTP default trace level value is returned. The default value is AFTP_LVL_NO_TRACING.

```
AFTP_ENTRY aftp_extract_trace_level(
    OUT AFTP_TRACE_LEVEL_TYPE AFTP_PTR trace_level,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

trace_level
The current setting of the trace level in AFTP. The constants from AFTP_LVL_NO_TRACING to AFTP_LVL_MAX_TRACE_LVL incrementally increase the amount of trace information.

AFTP_LVL_NO_TRACING

Writes no data to the trace log.

AFTP_LVL_API

Traces crossings of the API boundary.

AFTP_LVL_MAX_TRACE_LVL

Provides the maximum amount of trace information.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         rc;
    AFTP_TRACE_LEVEL_TYPE         trace_level;

    /*
     * There are no prerequisite calls for this call.
     */

    aftp_extract_trace_level(&trace_level, &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting trace level\n");
    }

}
```

Line Flows

None.

aftp_extract_userid

The **aftp_extract_userid** call extracts the user identifier specified for the connection to the AFTP server. If the CPI-C Extract_Conversation_Security_User_ID function is not supported, the AFTP default user identifier value is returned.

```
AFTP_ENTRY aftp_extract_userid(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR userid,
    IN AFTP_LENGTH_TYPE userid_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

userid
The buffer into which the user identifier used on the connection will be written.

Use the AFTP_USERID_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

userid_size
The size of the buffer into which the user identifier will be written.

returned_length
The actual length of the *userid* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 userid[AFTP_USERID_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the user ID for AFTP.
     */

    aftp_extract_userid(
        connection_id,
        userid,
        (AFTP_LENGTH_TYPE)sizeof(userid)-1,
        &returned_length,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting userid.\n");
    }

}
```

Line Flows

None.

aftp_extract_write_mode

The **aftp_extract_write_mode** call extracts the way that existing files are treated when a data transfer writes to them. If the aftp_set_write_mode call has not been invoked, the AFTP default write mode value is returned.

```
AFTP_ENTRY aftp_extract_write_mode(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_WRITE_MODE_TYPE AFTP_PTR write_mode,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

write_mode
The method used to write a file if a copy of the file already exists. If the file does not exist on the target, a new file is created.

AFTP_REPLACE

Transferred file replaces the existing file.

AFTP_APPEND

Transferred file is appended to the existing file.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;
    AFTP_WRITE_MODE_TYPE   write_mode;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Extract the file write mode for AFTP.
     */

    aftp_extract_write_mode(
        connection_id,
        &write_mode,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error extracting AFTP write mode.\n");
    }
}
```

Line Flows

None.

aftp_format_error

The **aftp_format_error** call retrieves the current AFTP error information to a text buffer. The AFTP return code for the current error must not be AFTP_RC_HANDLE_NOT_VALID. If the current status is AFTP_RC_OK and the **aftp_format_error** call is invoked, the *return_code* value output by this call is AFTP_RC_STATE_CHECK. The **aftp_format_error** call should only be invoked when an error has occurred.

```
AFTP_ENTRY aftp_format_error(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_DETAIL_LEVEL_TYPE detail_level,
    OUT unsigned char AFTP_PTR error_str,
    IN AFTP_LENGTH_TYPE error_str_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

detail_level
The detail in which the error string will describe the AFTP error. These values can be **OR**'ed together to retrieve specific sets of information. For example, to return the primary message and the error log information, specify (AFTP_DETAIL_RC | AFTP_DETAIL_LOG).

AFTP_DETAIL_RC

The AFTP return code, error category, index, and primary error message will be output.

AFTP_DETAIL_SECOND

The AFTP secondary error message will be output.

AFTP_DETAIL_LOG

The error logging information will be output.

AFTP_DETAIL_INFO

The informational message associated with the error will be output.

AFTP_DETAIL_ALL

All of the previous detail levels will be output in the error string.

error_str
The buffer into which the error information string will be written.

Use the AFTP_MESSAGE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

error_str_size
The size of the buffer into which the error information will be written.

returned_length
The actual length of the *error_str* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE        connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;
    unsigned char           error_string[AFTP_MESSAGE_SIZE+1];
    AFTP_LENGTH_TYPE        returned_length;


    /*
```

```
     * There are no specific prerequisite calls for this call,
     * but you must issue a call that returns an error return code
     */

    if (aftp_rc != AFTP_RC_OK) {
        /*
         * We had an AFTP error - so let's get
         * the description that corresponds to
         * the error.
         */

        aftp_format_error(
            connection_id,
            AFTP_DETAIL_ALL,
            error_string,
            (AFTP_LENGTH_TYPE)sizeof(error_string)-1,
            &returned_length,
            &aftp_rc);
    }
}
```

Line Flows

None.

aftp_get_data_type_string

The **aftp_get_data_type_string** call gets a string that corresponds to the input AFTP data type value. This string is available to allow all users of the AFTP API to have consistent strings for each data type. It is not necessary to create an AFTP connection object before issuing this call.

```
AFTP_ENTRY aftp_get_data_type_string(
    IN AFTP_DATA_TYPE_TYPE data_type,
    OUT unsigned char AFTP_PTR data_type_string,
    IN AFTP_LENGTH_TYPE data_type_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

data_type
An AFTP data type value.

AFTP_ASCII

Transfer files as text files in ASCII.

AFTP_BINARY

Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE

Use the data transfer type set in the .INI file. If no type is set in the .INI file, use AFTP_ASCII.

data_type_string
The buffer into which the data type string will be written.

Use the AFTP_DATA_TYPE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

data_type_size
The size of the buffer into which the data type string will be written.

returned_length
The actual length of the *data_type_string* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 data_type[AFTP_DATA_TYPE_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */
    /*
     * Get the data type string.
     */

    aftp_get_data_type_string(
        AFTP_ASCII,
        data_type,
        (AFTP_LENGTH_TYPE)sizeof(data_type)-1,
        &returned_length,
        &aftp_rc);
}
```

Line Flows

None.

aftp_get_date_mode_string

The **aftp_get_date_mode_string** call gets a string that corresponds to the input AFTP date mode value. This string is available to allow all users of the AFTP API to have consistent strings for each date mode type. It is not necessary to create an AFTP connection object before issuing this call.

```
AFTP_ENTRY aftp_get_date_mode_string(
    IN AFTP_DATE_MODE_TYPE date_mode,
    OUT unsigned char AFTP_PTR date_mode_string,
    IN AFTP_LENGTH_TYPE date_mode_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

date_mode
An AFTP date mode value.

AFTP_NEWDATE

Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE

Assign the time/date stamp of the source file.

date_mode_string
The buffer into which the date mode string will be written.

Use the AFTP_DATE_MODE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

date_mode_size
The size of the buffer into which the date mode string will be written.

returned_length
The actual length of the *date_mode_string* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 date_mode[AFTP_DATE_MODE_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Get the date mode string.
     */

    aftp_get_date_mode_string(
        AFTP_OLDDATE,
        date_mode,
        (AFTP_LENGTH_TYPE)sizeof(date_mode)-1,
        &returned_length,
        &aftp_rc);
}
```

Line Flows

None.

aftp_get_record_format_string

The **aftp_get_record_format_string** call gets a string that corresponds to the input AFTP record format value. This string is available to allow all users of the AFTP API to have consistent strings for each record format type. It is not necessary to create an AFTP connection object before issuing this call.

```
AFTP_ENTRY aftp_get_record_format_string(
    IN AFTP_RECORD_FORMAT_TYPE record_format,
    OUT unsigned char AFTP_PTR record_format_string,
    IN AFTP_LENGTH_TYPE record_format_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

record_format

An AFTP record format value.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FBA

Fixed length record, blocked, ASA print-control characters.

AFTP_FBM

Fixed length record, blocked, machine print-control codes.

AFTP_FBS

Fixed length record, blocked, standard.

AFTP_FBSA

Fixed length record, blocked, standard, ASA print-control characters.

AFTP_FBSM

Fixed length record, blocked, standard, machine print-control codes.

AFTP_FM

Fixed length record, unblocked, machine print-control codes.

AFTP_U

Undefined length record.

AFTP_UA

Undefined length record, ASA print-control characters.

AFTP_UM

Undefined length record, machine print-control codes.

record_format_string
The buffer into which the record format string will be written.

Use the AFTP_RECORD_FORMAT_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

record_format_size
The size of the buffer into which the record format string will be written.

returned_length
The actual length of the *record_format_string* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 recfm[AFTP_RECORD_FORMAT_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Get the record format string.
     */
```

```
    aftp_get_record_format_string(
        AFTP_F,
        recfm,
        (AFTP_LENGTH_TYPE)sizeof(recfm)-1,
        &returned_length,
        &aftp_rc);
}
```

Line Flows

None.

aftp_get_write_mode_string

The **aftp_get_write_mode_string** call gets a string that corresponds to the input AFTP write mode value. This string is available to allow all users of the AFTP API to have consistent strings for each write mode type. It is not necessary to create an AFTP connection object before issuing this call.

```
AFTP_ENTRY aftp_get_write_mode_string(
    IN AFTP_WRITE_MODE_TYPE write_mode,
    OUT unsigned char AFTP_PTR write_mode_string,
    IN AFTP_LENGTH_TYPE write_mode_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

write_mode
The method used to write a file if a copy of the file already exists. If the file does not exist on the target, a new file will be created.

AFTP_REPLACE

Transferred file will replace the existing file.

AFTP_APPEND

Transferred file will be appended to the existing file.

write_mode_string
The buffer into which the write mode string will be written.

Use the AFTP_WRITE_MODE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

write_mode_size
The size of the buffer into which the write mode string will be written.

returned_length
The actual length of the *write_mode_string* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 write_mode[AFTP_WRITE_MODE_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Get the write mode string.
     */

    aftp_get_write_mode_string(
        AFTP_REPLACE,
        write_mode
        (AFTP_LENGTH_TYPE)sizeof(write_mode)-1,
        &returned_length,
        &aftp_rc);
}
```

Line Flows

None.

aftp_load_ini_file

The **aftp_load_ini_file** call reads the AFTP initialization file into memory. This file includes information required to map file names on the current platform. When the AFTP initialization file is stored in memory, AFTP automatically consults the data it contains before proceeding with any operations. It is not necessary to create an AFTP connection object before issuing this call.

The name of the initialization file varies by operating system:

- **MVS:** DD:APPFTPI
- VM: AFTP INI
- Win32®: AFTP.INI

```
AFTP_ENTRY aftp_load_ini_file(
    IN unsigned char AFTP_PTR filename,
    IN AFTP_LENGTH_TYPE filename_size,
    IN unsigned char AFTP_PTR program_path,
    IN AFTP_LENGTH_TYPE path_size,
    OUT unsigned char AFTP_PTR error_string,
    IN AFTP_LENGTH_TYPE error_string_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

filename
The file name of the AFTP initialization file.

filename_size
The length of the *filename* parameter in bytes.

program_path
For Win32, the fully qualified file specification of the program that is running. The path from the file specification will be used to locate the AFTP initialization file.

For MVS or VM where this information is not available, provide a zero-length string (not a null string) for this parameter.

path_size
The length of the *program_path* parameter in bytes.

error_string
The buffer into which any error messages will be written during loading of the initialization file.

Use the AFTP_INI_MESSAGE_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

error_string_size
The size of the buffer into which the error information will be written.

returned_length
The actual size of the error information in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    static unsigned char AFTP_PTR init_file_name = "DD:APPFTPI";
    static unsigned char AFTP_PTR program_name = ""
    unsigned char                 error_string[AFTP_INI_MESSAGE_SIZE+1];
    AFTP_LENGTH_TYPE              returned_length;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Load the AFTP initialization file into memory.
```

```
      */
     aftp_load_ini_file(
         init_file_name,
         (AFTP_LENGTH_TYPE)strlen(init_file_name),
         program_name,
         (AFTP_LENGTH_TYPE)strlen(program_name),
         error_string,
         (AFTP_LENGTH_TYPE)sizeof(error_string),
         &returned_length,
         &aftp_rc);
     if (aftp_rc != AFTP_RC_OK {
         error_string[returned_length]='\0';
         printf(stderr, error_string);
     }
 }
```

Line Flows

None.

aftp_local_change_dir

The **aftp_local_change_dir** call changes the current working directory on the AFTP client. A connection to the AFTP server is not required before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_local_change_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The new directory name. The format of this name can be either the native syntax on the AFTP client or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The directory specified can be either an absolute or relative path name.

length
The length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /* The value used will vary based on platform:
     *   VM common naming:       directory = "/d"
     *   VM native naming:       directory = "/d"
     *   MVS PDS common naming:  directory = "/user.clist/"
     *   MVS PDS native naming:  directory = "'user.clist'"
     *   MVS data set prefix common:  directory = "/user.qual.a."
     *   MVS data set prefix native:  directory = "'user.qual.a.'"
     *   NT common naming:     directory = "/c:/nt"
     *   NT native naming:     directory = "c:\\nt"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/"; /* MVS */

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Specify the new current working directory name on the AFTP
     * client.
     */

    aftp_local_change_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)strlen(directory),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error changing AFTP directory.\n");
    }
}
```

Line Flows

None.

aftp_local_dir_close

The **aftp_local_dir_close** call cancels a directory listing that is in progress on the AFTP client or ends a directory listing on the AFTP client after a nonzero *no_more_entries* has been returned from an aftp_local_dir_read call. A connection to the AFTP server is not required before using this call. A directory listing on the AFTP client must be started by calling aftp_local_dir_open before making this call.

```
AFTP_ENTRY aftp_local_dir_close(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

See aftp_local_dir_read for a complete example showing the related calls **aftp_local_dir_open**, **aftp_local_dir_read**, and **aftp_local_dir_close**.

Line Flows

None.

aftp_local_dir_open

The **aftp_local_dir_open** call begins a directory listing and specifies the file search parameters on the AFTP client. The aftp_local_dir_read call is used to read individual directory entries. The aftp_local_dir_close call is used to end the directory listing. A connection to the AFTP server is not required before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_local_dir_open(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR filespec,
    IN AFTP_LENGTH_TYPE length,
    IN AFTP_FILE_TYPE_TYPE file_type,
    IN AFTP_INFO_LEVEL_TYPE info_level,
    OUT unsigned char AFTP_PTR path,
    IN AFTP_LENGTH_TYPE path_buffer_length,
    OUT AFTP_LENGTH_TYPE AFTP_PTR path_returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

filespec
The search string that the client uses to generate the directory listing. The files in the listing must match the search string. The format of this name can be either the native syntax on the AFTP client or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can be either an absolute or relative path name and can contain wildcard characters.

length
The length of the *filespec* parameter in bytes.

file_type
The type of information (directory names or file names) to be returned.

AFTP_FILE

Only file entries.

AFTP_DIRECTORY

Only directory entries.

AFTP_ALL_FILES

Both file and directory entries.

info_level
The level and format of information to be returned about each file or directory entry.

AFTP_NATIVE_NAMES

Native names without attributes.

AFTP_NATIVE_ATTRIBUTES

Native names and native file attributes.

path
The fully qualified directory name in which of all of the directory entries exist. The actual directory entries will be returned when the aftp_local_dir_read call is used. The path can be used along with the returned directory entry file name to create a fully qualified path name to use on another AFTP file call.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

path_buffer_length
The size in bytes of the buffer pointed to by the *path* parameter.

path_returned_length

The number of bytes returned in the *path* parameter.

return_code

The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

See aftp_local_dir_read for a complete example showing the related calls **aftp_local_dir_open**, **aftp_local_dir_read**, and **aftp_local_dir_close**.

Line Flows

None.

aftp_local_dir_read

The **aftp_local_dir_read** call gets an individual directory entry from the AFTP client, based on the search parameters specified on the aftp_local_dir_open call. A connection to the AFTP server is not required before using this call. The **aftp_local_dir_open** call must be issued before listing the directory entries.

```
AFTP_ENTRY aftp_local_dir_read(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR dir_entry,
    IN AFTP_LENGTH_TYPE dir_entry_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_BOOLEAN_TYPE AFTP_PTR no_more_entries,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

dir_entry
Pointer to a buffer into which the procedure will write the directory entry.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

dir_entry_size
The size in bytes of the *dir_entry* buffer.

returned_length
The number of bytes returned in the *dir_entry* parameter.

no_more_entries
Whether or not an entry was returned on this call.

A value of zero indicates that there are more directory entries and that an entry was returned on this call.

A nonzero value indicates that there are no more directory entries and that no entry was returned on this call. The *returned_length* parameter is set to zero. Subsequent calls to aftp_local_dir_read will also result in *no_more_entries* being nonzero. To end the directory listing, your next call should be aftp_local_dir_close.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;
    unsigned char                 dir_entry[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE              dir_entry_length;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filespec="/a/foo*"
     *   VM native naming:       filespec="foo*.*.a"
     *   MVS PDS common naming:  filespec="/user.clist/foo*"
     *   MVS PDS native naming:  filespec="'user.clist(foo*)'"
     *   MVS sequential common:  filespec="/user.qual*.a*.**"
     *   MVS sequential native:  filespec="'user.qual*.a*.**'"
     */
    static unsigned char AFTP_PTR filespec = "/user.clist/foo*";

    unsigned char                 path[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE              path_length;
    AFTP_BOOLEAN_TYPE             no_more_entries;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
```

```
     */

    /*
     * Open a new directory listing on the AFTP client. Both files and
     * directory names will be listed along with their attributes.
     */

    aftp_local_dir_open(
        connection_id,
        filespec,
        (AFTP_LENGTH_TYPE)strlen(filespec),
        AFTP_DIRECTORY | AFTP_FILE,
        AFTP_NATIVE_ATTRIBUTES,
        path,
        (AFTP_LENGTH_TYPE)sizeof(path)-1,
        &path_length,
        &aftp_rc);

    if (aftp_rc == AFTP_RC_OK) {
        path[path_length] = '\0';

        printf("Directory listing of %s.", path);

        do {
            /*
             * Read one directory entry from the AFTP client
             */

            aftp_local_dir_read(
                connection_id,
                dir_entry,
                (AFTP_LENGTH_TYPE)sizeof(dir_entry)-1,
                &dir_entry_length,
                &no_more_entries,
                &aftp_rc);

            if (aftp_rc == AFTP_RC_OK && no_more_entries == 0) {
                dir_entry[dir_entry_length] = '\0';
                printf("Local file: %s\n", dir_entry);
            }
            /*
             * Loop until we either run out of directory
             * entries or an error occurs.
             */

        } while (aftp_rc == AFTP_RC_OK && no_more_entries == 0);

        /*
         * Terminate the directory listing by executing
         * a close.
         */

        aftp_local_dir_close(connection_id, &aftp_rc);
        if (aftp_rc != AFTP_RC_OK) {
            fprintf(
                stderr,
                "Error closing local AFTP directory.\n");
        }
    }
    else {
        fprintf(stderr, "Error opening local AFTP directory.\n");
    }
}
```

Line Flows

None.

aftp_local_query_current_dir

The **aftp_local_query_current_dir** call queries the current working directory on the AFTP client. A connection to the AFTP server is not required before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_local_query_current_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE directory_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The buffer into which the current working directory on the AFTP client will be written.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

directory_size
The size in bytes of the *directory* buffer.

returned_length
The actual length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE        connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;
    unsigned char           directory[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE        length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Query the current working directory on the
     * AFTP client.
     */

    aftp_local_query_current_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)sizeof(directory)-1,
        &length,
        &aftp_rc);

    directory[length] = '\0';

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error in query of local current directory.\n");
    }
}
```

Line Flows

None.

aftp_query_bytes_transferred

The **aftp_query_bytes_transferred** call queries the total number of bytes transferred after either an aftp_send_file or aftp_receive_file call has completed. The number of bytes transferred is valid only after a file transfer operation has completed. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_query_bytes_transferred(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT AFTP_LENGTH_TYPE AFTP_PTR bytes_transferred,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

bytes_transferred
The number of bytes of data transferred during the last send or receive file operation.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE        connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;
    AFTP_LENGTH_TYPE        number_bytes;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     *    completed a send or receive, use:  aftp_send()
     *    or aftp_receive()
     */

    aftp_query_bytes_transferred(
        connection_id,
        &number_bytes,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error getting number bytes transferred.\n");
    } else {
        fprintf(stdout, "Number of bytes %d.\n", (int)number_bytes);
    }
}
```

Line Flows

None.

aftp_query_current_dir

The **aftp_query_current_dir** call queries the current directory on the AFTP server. A connection to the AFTP server must be established before using this call.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_query_current_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE directory_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The buffer into which the current working directory on the AFTP server will be written.

Use the AFTP_FILE_NAME_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

directory_size
The size in bytes of the *directory* buffer.

returned_length
The actual length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            directory[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    /*
     * Query the current working directory on the AFTP server.
     */

    aftp_query_current_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)sizeof(directory)-1,
        &length,
        &aftp_rc);


    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error in query of current directory.\n");
    }
}
```

Line Flows

The request for the directory name is sent to the AFTP server and the call waits for a response indicating the success or failure of the query current working directory operation. The directory name of the current working directory on the AFTP server is sent as the response if the query was successful.

aftp_query_local_system_info

The **aftp_query_local_system_info** call gets information about the AFTP client and the computer it is running on. A connection to the AFTP server is not required before using this call.

```
AFTP_ENTRY aftp_query_local_system_info(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR system_info,
    IN AFTP_LENGTH_TYPE system_info_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

system_info
Buffer to store a text string describing the operating system and AFTP client version.

Use the AFTP_SYSTEM_INFO_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

system_info_size
Size in bytes of the *system_info* parameter.

returned_length
Number of bytes stored in the *system_info* parameter.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            system_info[AFTP_FILE_NAME_SIZE+1];
    AFTP_LENGTH_TYPE         system_info_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Query the AFTP client computer for more information.
     */

    aftp_query_local_system_info(
        connection_id,
        system_info,
        (AFTP_LENGTH_TYPE)sizeof(system_info)-1,
        &system_info_length,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error querying AFTP client system.\n");
    }
}
```

Line Flows

None.

aftp_query_local_version

The **aftp_query_local_version** call queries the AFTP version number on the AFTP client computer. A connection to the AFTP server is not required before using this call.

```
AFTP_ENTRY aftp_query_local_version(
    OUT AFTP_VERSION_TYPE AFTP_PTR major_version,
    OUT AFTP_VERSION_TYPE AFTP_PTR minor_version,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

major_version
The major version number of the AFTP code on the client computer. In version 5.4, the major version number is 5.
minor_version
The minor version number of the AFTP code on the client computer. In version 5.4, the minor version number is 4.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE     aftp_rc;
    AFTP_VERSION_TYPE         major_version;
    AFTP_VERSION_TYPE         minor_version;

    /*
     * There are no prerequisite calls for this call.
     */

    /*
     * Query the AFTP version number on the
     * AFTP client computer.
     */

    aftp_query_local_version(
        &major_version,
        &minor_version,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error in query of local version.\n");
    }
}
```

Line Flows

None.

aftp_query_system_info

The **aftp_query_system_info** call gets information about the AFTP server and the computer it is running on. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_query_system_info(
    IN AFTP_HANDLE_TYPE connection_id,
    OUT unsigned char AFTP_PTR system_info,
    IN AFTP_LENGTH_TYPE system_info_size,
    OUT AFTP_LENGTH_TYPE AFTP_PTR returned_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

system_info
Buffer to store a text string describing the operating system and AFTP server version.

Use the AFTP_SYSTEM_INFO_SIZE constant to define the length of this buffer. Add 1 to the size if you want to be able to add a null terminator to the text in the buffer.

system_info_size
Size in bytes of the *system_info* parameter.

returned_length
Number of bytes stored in the *system_info* parameter.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;
    unsigned char            system_info[AFTP_SYSTEM_INFO_SIZE+1];
    AFTP_LENGTH_TYPE         system_info_length;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    /*
     * Query the AFTP server computer for more information.
     */

    aftp_query_system_info(
        connection_id,
        system_info,
        (AFTP_LENGTH_TYPE)sizeof(system_info)-1,
        &system_info_length,
        &aftp_rc);

    system_info[system_info_length] = '\0';

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error querying AFTP server system.\n");
    }
}
```

Line Flows

The request for the system information is sent to the AFTP server and the call waits for a response indicating the success or failure of the query system information operation. The system information of the AFTP server computer is sent as the response if the query was successful.

aftp_receive_file

The **aftp_receive_file** call receives a single file from the AFTP server. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_receive_file(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR local_file,
    IN AFTP_LENGTH_TYPE local_file_length,
    IN unsigned char AFTP_PTR remote_file,
    IN AFTP_LENGTH_TYPE remote_file_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

local_file
The name given to the file received on the AFTP client. The format of this name can be either the native syntax on the AFTP client or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can contain either an absolute or relative path name.

local_file_length
The length of the *local_file* parameter in bytes.

remote_file
The name of the file sent from the AFTP server. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can contain either an absolute or relative path name.

remote_file_length
The length of the *remote_file* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE                    connection_id;
    AFTP_RETURN_CODE_TYPE               aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filespec="/a/myfile.dat"
     *   VM native naming:       filespec="myfile.dat.a"
     *   MVS PDS common naming:  filespec="/user.mypds/myfile"
     *   MVS PDS native naming:  filespec="'user.mypds(myfile)'"
     *   MVS sequential common:  filespec="/user.qual.myfile"
     *   MVS sequential native:  filespec="'user.qual.myfile'"
     */
    static unsigned char AFTP_PTR local_file = "/user.mypos/myfile";
    /* MVS */
        static unsigned char AFTP_PTR remote_file = "/a/myfile.dat";
    /* VM */

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    aftp_receive_file(
        connection_id,
        local_file,
        (AFTP_LENGTH_TYPE)strlen(local_file),
        remote_file,
        (AFTP_LENGTH_TYPE)strlen(remote_file),
```

```
          &aftp_rc);

      if (aftp_rc != AFTP_RC_OK) {
          fprintf(stderr, "Error receiving AFTP file.\n");
      }
  }
```

Line Flows

The request to receive the file is sent to the AFTP server. A send file indicator is returned to the AFTP client. All records of the file are then sent from the AFTP server to the AFTP client.

aftp_remove_dir

The **aftp_remove_dir** call removes a directory from the AFTP server. A connection to the AFTP server must be established before using this call.

Platform differences are as follows:

- On VM, this call is not supported. If issued, the call fails with return code AFTP_RC_FAIL_NO_RETRY.
- On MVS, partitioned data sets act as the directory structure. This call deletes a partitioned data set with the name specified.

See AFTP File and Directory Concepts for details on how the directory concept is handled for supported operating systems.

```
AFTP_ENTRY aftp_remove_dir(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR directory,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

directory
The directory to be removed. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The directory specified can be either an absolute or relative path name.

length
The length of the *directory* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE                    connection_id;
    AFTP_RETURN_CODE_TYPE               aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM  not supported
     *   MVS PDS common naming:  directory="/user.clist/"
     *   MVS PDS native naming:  directory="'user.clist'"
     */
    static unsigned char AFTP_PTR directory = "/user.clist/";

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    aftp_remove_dir(
        connection_id,
        directory,
        (AFTP_LENGTH_TYPE)strlen(directory),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error removing AFTP directory.\n");
    }

}
```

Line Flows

The remove directory request and the directory name to remove are sent to the AFTP server and the call waits for a response indicating the success or failure of the remove directory operation.

aftp_rename

The **aftp_rename** call renames a file on the AFTP server. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_rename(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR oldfile,
    IN AFTP_LENGTH_TYPE oldlength,
    IN unsigned char AFTP_PTR newfile,
    IN AFTP_LENGTH_TYPE newlength,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

oldfile
The name of the file to be renamed.

The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can be either an absolute or relative path name.

oldlength
The length in bytes of the *oldfile* parameter.

newfile
The new name of the file.

The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can be either an absolute or relative path name.

newlength
The length in bytes of the *newfile* parameter.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       newfile="/a/foo.file"
     *   VM native naming:       newfile="foo.file.a"
     *   MVS PDS common naming:  newfile="/user.clist/foo"
     *   MVS PDS native naming:  newfile="'user.clist(foo)'"
     *   MVS sequential common:  newfile="/user.qual.a.foo"
     *   MVS sequential native:  newfile="'user.qual.a.foo'"
     */
    static unsigned char AFTP_PTR newfile = "/user.clist/foo";
    static unsigned char AFTP_PTR oldfile = "/user.clist/abc";

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    aftp_rename(
        connection_id,
        oldfile,
        (AFTP_LENGTH_TYPE)strlen(oldfile),
        newfile,
        (AFTP_LENGTH_TYPE)strlen(newfile),
```

```
            &aftp_rc);

        if (aftp_rc != AFTP_RC_OK) {
            fprintf(stderr, "Error renaming AFTP file.\n");
        }

    }
```

Line Flows

The rename request and the old and new file names are sent to the AFTP server and the call waits for a response indicating the success or failure of the rename operation.

aftp_send_file

The **aftp_send_file** call sends a single file to the AFTP server. A connection to the AFTP server must be established before using this call.

```
AFTP_ENTRY aftp_send_file(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR local_file,
    IN AFTP_LENGTH_TYPE local_file_length,
    IN unsigned char AFTP_PTR remote_file,
    IN AFTP_LENGTH_TYPE remote_file_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

local_file
The name of the file sent from the AFTP client. The format of this name can be either the native syntax on the AFTP client or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can contain either an absolute or relative path name.

local_file_length
The length of the *local_file* parameter in bytes.

remote_file
The name given to the file received on the AFTP server. The format of this name can be either the native syntax on the AFTP server or the AFTP common naming convention described in the *APPC Application Suite User's Guide*. The file specified can contain either an absolute or relative path name.

remote_file_length
The length of the *remote_file* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE                    connection_id;
    AFTP_RETURN_CODE_TYPE               aftp_rc;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filespec="/a/myfile.dat"
     *   VM native naming:       filespec="myfile.dat.a"
     *   MVS PDS common naming:  filespec="/user.mypds/myfile"
     *   MVS PDS native naming:  filespec="'user.mypds(myfile)'"
     *   MVS sequential common:  filespec="/user.qual.myfile"
     *   MVS sequential native:  filespec="'user.qual.myfile'"
     */
    static unsigned char AFTP_PTR local_file = "/user.mypos/myfile";
    /* MVS */
        static unsigned char AFTP_PTR remote_file = "/a/myfile.dat";
    /* VM */

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:         aftp_create()
     *    a connection to server, use:  aftp_connect()
     */

    aftp_send_file(
        connection_id,
        local_file,
        (AFTP_LENGTH_TYPE)strlen(local_file),
        remote_file,
        (AFTP_LENGTH_TYPE)strlen(remote_file),
```

```
            &aftp_rc);

        if (aftp_rc != AFTP_RC_OK) {
            fprintf(stderr, "Error sending AFTP file.\n");
        }
    }
```

Line Flows

The send file request is sent to the AFTP server, immediately followed by all records of the files. The call waits for a response indicating the success or failure of the send file operation.

aftp_set_allocation_size

The **aftp_set_allocation_size** call sets the AFTP file allocation size. A connection to the AFTP server is not required before using this call. The file allocation size can be changed at any time.

```
AFTP_ENTRY aftp_set_allocation_size(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_ALLOCATION_SIZE_TYPE allocation_size,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
allocation_size
The allocation size in bytes to set for the AFTP file. The default allocation size value is zero.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE    aftp_rc;

   /*
    * Before issuing the example call, you must have:
    *    a connection_id, use:  aftp_create()
    */

   /*
    * Set the file allocation size for AFTP file
    * transfers.
    */

    aftp_set_allocation_size(
        connection_id,
        500,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP allocation size.\n");
    }
}
```

Line Flows

None.

aftp_set_block_size

The **aftp_set_block_size** call sets the file block size. A connection to the AFTP server is not required before using this call. The file block size can be changed at any time.

```
AFTP_ENTRY aftp_set_block_size(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_BLOCK_SIZE block_size,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
block_size
The AFTP file block size in bytes. The default block size value is zero.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE         connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Set the file block size for AFTP file
     * transfers.
     */

    aftp_set_block_size(
        connection_id,
        512,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP block size.\n");
    }

}
```

Line Flows

None.

aftp_set_data_type

The **aftp_set_data_type** call sets the data type for file transfers. A connection to the AFTP server is not required before using this call. The data type can be changed at any time.

```
AFTP_ENTRY aftp_set_data_type(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_DATA_TYPE_TYPE data_type,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

data_type
The data type to be used for subsequent data transfers.

AFTP_ASCII

Transfer files as text files in ASCII.

AFTP_BINARY

Transfer files as a binary sequence of bytes without translation.

AFTP_DEFAULT_DATA_TYPE

Use the data transfer type set in the .INI file. If no type is set in the .INI file, use AFTP_ASCII.

This is the default setting.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Set the data type for AFTP file
     * transfers.
     */

    aftp_set_data_type(
        connection_id,
        AFTP_BINARY,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP data type.\n");
    }

}
```

Line Flows

None.

aftp_set_date_mode

The **aftp_set_date_mode** call sets the way file dates are handled during data transfer. A connection to the AFTP server is not required before using this call. The date mode can be changed at any time.

```
AFTP_ENTRY aftp_set_date_mode(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_DATE_MODE_TYPE date_mode,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

date_mode
Specifies the way file dates are handled during data transfer.

AFTP_NEWDATE

Assign the time/date stamp of the time of transfer.

AFTP_OLDDATE

Assign the time/date stamp of the source file. This is the default.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE        connection_id;
    AFTP_RETURN_CODE_TYPE   aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Set the date mode for AFTP file
     * transfers.
     */

    aftp_set_date_mode(
        connection_id,
        AFTP_OLDDATE,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP date mode.\n");
    }

}
```

Line Flows

None.

aftp_set_destination

The **aftp_set_destination** call specifies the destination of the AFTP server. This call must be issued before establishing a connection to the AFTP server. After a connection is established, the destination cannot be changed.

```
AFTP_ENTRY aftp_set_destination(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR destination,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

destination
Identifies the location of the AFTP server. This parameter can be either a symbolic destination name or a partner LU name.

See the *APPC Application Suite User's Guide* for information about specifying destinations in the APPC Application Suite.

length
The length of the *destination* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE             connection_id;
    static unsigned char AFTP_PTR destination = "NETWORK.SERVER";
    AFTP_RETURN_CODE_TYPE        aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the partner we want to communicate with - who will
     * be running the AFTP server.
     */

    aftp_set_destination(
        connection_id,
        destination,
        (AFTP_LENGTH_TYPE)strlen(destination),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP destination.\n");
    }
}
```

Line Flows

None.

aftp_set_mode_name

The **aftp_set_mode_name** call specifies the mode name for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. When a connection is open, the mode name cannot be changed.

```
AFTP_ENTRY aftp_set_mode_name(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR mode_name,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

mode_name
Specifies the mode name to be used on the connection. The default is #BATCH. The mode name must be from 1 through 8 bytes long.

length
The length of the *mode_name* parameter in bytes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    static unsigned char AFTP_PTR mode_name = "#INTER";
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the mode name for the AFTP connection.
     */

    aftp_set_mode_name(
        connection_id,
        mode_name,
        (AFTP_LENGTH_TYPE)strlen(mode_name),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP mode name.\n");
    }

}
```

Line Flows

None.

aftp_set_password

The **aftp_set_password** call specifies the password for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. When a connection is open, the password cannot be changed. If a password is set, a user identifier also must be set using aftp_set_userid before connecting to the AFTP server. Use of this call sets the security type to AFTP_SECURITY_PROGRAM.

```
AFTP_ENTRY aftp_set_password(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR password,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
password
The password to be used on the connection. The password can be from 1 through 8 bytes long.
length
The length of the *password* parameter in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    static unsigned char AFTP_PTR password = "MYPASS";
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the password for the AFTP connection.
     */

    aftp_set_password(
        connection_id,
        password,
        (AFTP_LENGTH_TYPE)strlen(password),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP password.\n");
    }

}
```

Line Flows

None.

aftp_set_record_format

The **aftp_set_record_format** call sets the record format for the data transfer. A connection to the AFTP server is not required before using this call. The record format can be changed at any time.

```
AFTP_ENTRY aftp_set_record_format(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_RECORD_FORMAT_TYPE record_format,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

record_format
The record format used for file transfer.

AFTP_DEFAULT_RECORD_FORMAT

Specifies that the system on which the file will be written should use its own default setting for record format. This is the initial setting.

AFTP_V

Variable length record, unblocked.

AFTP_VA

Variable length record, unblocked, ASA print-control characters.

AFTP_VB

Variable length record, blocked.

AFTP_VBA

Variable length record, blocked, ASA print-control characters.

AFTP_VBM

Variable length record, blocked, machine print-control codes.

AFTP_VBS

Variable length record, blocked, spanned.

AFTP_VBSA

Variable length record, blocked, spanned, ASA print-control characters.

AFTP_VBSM

Variable length record, blocked, spanned, machine print-control codes.

AFTP_VM

Variable length record, unblocked, machine print-control codes.

AFTP_VS

Variable length record, unblocked, spanned.

AFTP_VSA

Variable length record, unblocked, spanned, ASA print-control characters.

AFTP_VSM

Variable length record, unblocked, spanned, machine print-control codes.

AFTP_F

Fixed length record, unblocked.

AFTP_FA

Fixed length record, unblocked, ASA print-control characters.

AFTP_FB

Fixed length record, blocked.

AFTP_FBA

Fixed length record, blocked, ASA print-control characters.

AFTP_FBM

Fixed length record, blocked, machine print-control codes.

AFTP_FBS

Fixed length record, blocked, standard.

AFTP_FBSA

Fixed length record, blocked, standard, ASA print-control characters.

AFTP_FBSM

Fixed length record, blocked, standard, machine print-control codes.

AFTP_FM

Fixed length record, unblocked, machine print-control codes.

AFTP_U

Undefined length record.

AFTP_UA

Undefined length record, ASA print-control characters.

AFTP_UM

Undefined length record, machine print-control codes.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /*
     * Before issuing the example call, you must have:
     * a connection_id, use:  aftp_create()
     */

    /*
     * Set the record format value for the file transfer.
     */

    aftp_set_record_format(
        connection_id,
        AFTP_VSA,
        &aftp_rc);
    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP record format.\n");
    }

}
```

Line Flows

None.

aftp_set_record_length

The **aftp_set_record_length** call sets the record length for fixed length records, or the maximum possible record length for variable length records used for data transfer. A connection to the AFTP server is not required before using this call. The record length can be changed at any time.

```
AFTP_ENTRY aftp_set_record_length(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_RECORD_LENGTH_TYPE record_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
record_length
The record length for the data transfer specified in bytes. The default value is zero.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Set the record length for the file transfer.
     */

    aftp_set_record_length(
        connection_id,
        64,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP record length.\n");
    }

}
```

Line Flows

None.

aftp_set_security_type

The **aftp_set_security_type** call specifies the type of APPC conversation security to be used. This call can only be invoked prior to the establishment of a connection to the AFTP server. When a connection is open, the APPC security type cannot be changed. If AFTP_SECURITY_PROGRAM is used for the security type, a user identifier and password must also be set using aftp_set_userid and aftp_set_password before connecting to the AFTP server.

```
AFTP_ENTRY aftp_set_security_type(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_SECURITY_TYPE security_type,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

security_type
The security to be used when connecting to the AFTP server.

AFTP_SECURITY_NONE

No APPC conversation security is used. This is the default unless CPI-C side information is set otherwise.

AFTP_SECURITY_SAME

The local security information determined at logon time will be transferred to the AFTP server.

AFTP_SECURITY_PROGRAM

A user identifier and password will be sent to be verified by the AFTP server. You must use the aftp_set_userid and aftp_set_password calls with this security type, or the connection attempt will fail.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the APPC conversation security type for the
     * AFTP connection.
     */

    aftp_set_security_type(
        connection_id,
        AFTP_SECURITY_SAME,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP security type.\n");
    }
}
```

Line Flows

None.

aftp_set_tp_name

The **aftp_set_tp_name** call specifies the transaction program (TP) name of the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. When a connection is open, the TP name cannot be changed. The AFTP API defaults the TP name on the server to be AFTPD. This call is not necessary unless you want to experiment with a server that might not have the same behavior as AFTPD.

```
AFTP_ENTRY aftp_set_tp_name(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR tp_name,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
tp_name
The TP name of the AFTP server. The TP name can be from 1 through 64 bytes long. The default TP name is AFTPD.
length
The length of the *tp_name* parameter in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    static unsigned char AFTP_PTR tp_name = "AFTPME";
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the TP name for the AFTP server.
     */

    aftp_set_tp_name(
        connection_id,
        tp_name,
        (AFTP_LENGTH_TYPE)strlen(tp_name),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP TP name.\n");
    }
}
```

Line Flows

None.

aftp_set_trace_filename

The **aftp_set_trace_filename** call sets the name of the file to which trace information will be written. The default value for the trace level is AFTP_LVL_NO_TRACING. If trace is turned on by the aftp_set_trace_level call and the **aftp_set_trace_filename** call is not issued, the trace file generated is:

- **On MVS:** DD:SYSOUT
- On VM: ASUITE TRC
- On Win32®: ASUITE.TRC

```
AFTP_ENTRY aftp_set_trace_filename(
    IN unsigned char AFTP_PTR filename,
    IN AFTP_LENGTH_TYPE filename_length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

filename
The name of the file to be used for trace output.
filename_length
The length of the *filename* parameter in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_RETURN_CODE_TYPE      rc;

    /* The value used for filespec will vary based on platform:
     *   VM common naming:       filename="/a/aftp.trace"
     *   VM native naming:       filename="aftp.trace.a"
     *   MVS PDS common naming:  filename="/user.clist/aftptrac"
     *   MVS PDS native naming:  filename="'user.clist(aftptrac)'"
     *   MVS sequential common:  filename="/user.qual.a.aftptrac"
     *   MVS sequential native:  filename="'user.qual.a.aftptrac'"
     */
    static unsigned char AFTP_PTR filename = "/user.clist/aftptrac";

    /*
     * There are no prerequisite calls for this call.
     */

    aftp_set_trace_filename(
        filename,
        (AFTP_LENGTH_TYPE)strlen(filename),
        &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting tracing filename\n");
    }
}
```

Line Flows

None.

aftp_set_trace_level

The **aftp_set_trace_level** call sets the level of tracing to use for AFTP activities. The new trace level will take effect immediately upon making this call. The trace output is captured in the file specified in the aftp_set_trace_filename call that must be issued before this call.

```
AFTP_ENTRY aftp_set_trace_level(
    IN AFTP_TRACE_LEVEL_TYPE trace_level,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

trace_level

The amount of trace information to be generated. The constants from AFTP_LVL_NO_TRACING to AFTP_LVL_MAX_TRACE_LVL incrementally increase the amount of trace information.

AFTP_LVL_NO_TRACING

Writes no data to the trace log.

AFTP_LVL_API

Traces crossings of the API boundary.

AFTP_LVL_MAX_TRACE_LVL

Provides the maximum amount of trace information.

Other trace levels are reserved for diagnosing problems with the assistance of vendor support.

return_code

The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{

    /*
     * The following calls must be issued here:
     *    aftp_set_trace_filename()
     */

    /*
     * Turn on the tracing.
     */

    aftp_set_trace_level(trace_level, &rc);

    if (rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting the trace level\n");
    }
}
```

Line Flows

None.

aftp_set_userid

The **aftp_set_userid** call specifies the user identifier for the connection to the AFTP server. This call can only be invoked prior to the establishment of a connection to the AFTP server. When a connection is open, the user identifier cannot be changed. If a user identifier is set, a password also must be set using aftp_set_password before connecting to the AFTP server. Use of this call sets the security type to AFTP_SECURITY_PROGRAM.

```
AFTP_ENTRY aftp_set_userid(
    IN AFTP_HANDLE_TYPE connection_id,
    IN unsigned char AFTP_PTR userid,
    IN AFTP_LENGTH_TYPE length,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.
userid
The user identifier to be used on the connection. The user identifier can be from 1 through 8 bytes long.
length
The length of the *userid* parameter in bytes.
return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE              connection_id;
    static unsigned char AFTP_PTR userid = "LBONANNO";
    AFTP_RETURN_CODE_TYPE         aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     *
     * You cannot have an open connection.
     */

    /*
     * Set the user ID for the AFTP connection.
     */

    aftp_set_userid(
        connection_id,
        userid,
        (AFTP_LENGTH_TYPE)strlen(userid),
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting user ID.\n");
    }
}
```

Line Flows

None.

aftp_set_write_mode

The **aftp_set_write_mode** call sets the way existing files will be handled during data transfer. A connection to the AFTP server is not required before using this call. The write mode can be changed at any time.

```
AFTP_ENTRY aftp_set_write_mode(
    IN AFTP_HANDLE_TYPE connection_id,
    IN AFTP_WRITE_MODE_TYPE write_mode,
    OUT AFTP_RETURN_CODE_TYPE AFTP_PTR return_code
);
```

Parameters

connection_id
An AFTP connection object originally created with aftp_create.

write_mode
The method used to write a file if a copy of the file already exists. If the file does not exist on the target, a new file is created.

AFTP_REPLACE

Transferred file will replace the existing file. This is the default.

AFTP_APPEND

Transferred file will be appended to the existing file.

return_code
The return code issued for this function. See AFTP Return Codes for the list of possible return codes.

Example

```
{
    AFTP_HANDLE_TYPE       connection_id;
    AFTP_RETURN_CODE_TYPE  aftp_rc;

    /*
     * Before issuing the example call, you must have:
     *    a connection_id, use:  aftp_create()
     */

    /*
     * Set the write mode for the AFTP
     * file transfer.
     */

    aftp_set_write_mode(
        connection_id,
        AFTP_REPLACE,
        &aftp_rc);

    if (aftp_rc != AFTP_RC_OK) {
        fprintf(stderr, "Error setting AFTP write mode.\n");
    }
}
```

Line Flows

None.

AFTP Return Codes

These are the possible return codes that can be issued for each AFTP API call.

AFTP_RC_BUFFER_TOO_SMALL
: The buffer supplied by the caller for output data was too small to hold the data.

AFTP_RC_COMM_CONFIG_LOCAL
: The call failed due to a local configuration error. Communications will fail until the configuration problem is resolved.

AFTP_RC_COMM_CONFIG_REMOTE
: The call failed due to a remote configuration error. Communications will fail until the configuration problem is resolved.

AFTP_RC_COMM_FAIL_NO_RETRY
: The call failed due to a communications problem. The call will not successfully complete using the current parameters.

AFTP_RC_COMM_FAIL_RETRY
: The call failed due to a communications problem. The call might successfully complete if tried again.

AFTP_RC_FAIL_FATAL
: A serious system error has occurred. No calls can complete successfully.

AFTP_RC_FAIL_INPUT_ERROR
: The call might successfully complete after new input parameters are supplied.

AFTP_RC_FAIL_NO_RETRY
: The call will not successfully complete using the current parameters.

AFTP_RC_FAIL_RETRY
: The call might successfully complete if tried again.

AFTP_RC_HANDLE_NOT_VALID
: The call failed because the AFTP connection object passed into the AFTP API was not valid.

AFTP_RC_OK
: The call completed successfully.

AFTP_RC_PARAMETER_CHECK
: The call failed due to an error in one of the parameters passed into the AFTP API.

AFTP_RC_PROGRAM_INTERNAL_ERROR
: The call failed due to a programming error.

AFTP_RC_SECURITY_NOT_VALID
: The call failed because of APPC security.

AFTP_RC_STATE_CHECK
: The call failed because the current AFTP API call was made when AFTP was not in the state required for the call. For example, you will get a state check error if you try to use aftp_format_error when no error has occurred.

Entry Point Mappings

The following table shows the C function entry point associated with each AFTP API call.

Call name	Entry point
aftp_change_dir	ftcd
aftp_close	ftclose
aftp_connect	ftconn
aftp_create	ftcreate
aftp_create_dir	ftcrtdir
aftp_delete	ftdel
aftp_destroy	ftdestry
aftp_dir_close	ftdircls
aftp_dir_open	ftdiropn
aftp_dir_read	ftdirrd
aftp_extract_allocation_size	fteas
aftp_extract_block_size	ftebs
aftp_extract_data_type	ftedt
aftp_extract_date_mode	ftedm
aftp_extract_destination	ftedst
aftp_extract_mode_name	ftemn
aftp_extract_partner_lu_name	fteplu
aftp_extract_password	ftepw
aftp_extract_record_format	fterf
aftp_extract_record_length	fterl
aftp_extract_security_type	ftest
aftp_extract_tp_name	ftetpn
aftp_extract_trace_level	ftetl
aftp_extract_userid	fteui
aftp_extract_write_mode	ftewm
aftp_format_error	ftfe
aftp_get_data_type_string	ftgdts
aftp_get_date_mode_string	ftgdms
aftp_get_record_format_string	ftgrfs
aftp_get_write_mode_string	ftgwms
aftp_load_ini_file	ftlif
aftp_local_change_dir	ftlcd
aftp_local_dir_close	ftldc
aftp_local_dir_open	ftldo
aftp_local_dir_read	ftldr
aftp_local_query_current_dir	ftlqcd
aftp_query_bytes_transferred	ftqbt
aftp_query_current_dir	ftqcd
aftp_query_local_system_info	ftqlsi
aftp_query_local_version	ftqlv
aftp_query_system_info	ftqsys
aftp_receive_file	ftrecv
aftp_remove_dir	ftrd
aftp_rename	ftren
aftp_send_file	ftsend
aftp_set_allocation_size	ftsas
aftp_set_block_size	ftsbs
aftp_set_data_type	ftsdt
aftp_set_date_mode	ftsdm

aftp_set_destination	ftsdest
aftp_set_mode_name	ftsmn
aftp_set_password	ftsp
aftp_set_record_format	ftsrp
aftp_set_record_length	ftsrl
aftp_set_security_type	ftsst
aftp_set_tp_name	ftstp
aftp_set_trace_filename	ftstf
aftp_set_trace_level	ftstl
aftp_set_userid	ftsu
aftp_set_write_mode	ftswm

AFTP Sample Applications

This sample program shows a simple exercise of using the AFTP programming interface. It gets a single file from a remote machine. The user must know the machine name and the file name. Comments are inserted in bold text throughout the sample.

To link this sample code on Windows 2000, Windows NT, Windows 98, or Windows 95

1. The AFTP application must be statically linked with the AFTPAPI.LIB import library supplied as part of the Microsoft® Host Integration Server 2000 SDK or the earlier SNA Server 4.0 SDK.
2. Include the appropriate AFTPAPI.DLL with your application when it is installed on the target machine if this DLL is not already installed.

Note that Host Integration Server 2000 does not support the DEC Alpha and the Host Integration Server SDK does not include the DEC Alpha version of the AFTPAPI.LIB and AFTPAPI.DLL files. The DEC Alpha versions of these files are included as part of the earlier SNA Server 4.0 SDK.

System include files:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

AFTP API include file:

```
#include "aftpapi.h"
```

Note If you want to use the header file as it was shipped, change the file name in this include statement to APPFFTP.H. Otherwise, rename the member APPFFTP.H to AFTPAPI.H for consistency with other platforms.

```
int main(int argc, char *argv[])
{
    AFTP_HANDLE_TYPE connection_id;    /* connection id          */
    AFTP_RETURN_CODE_TYPE aftp_rc;     /* return code            */
    AFTP_SECURITY_TYPE sec_type;       /* security type          */
    unsigned char * LU_name;           /* partner LU name        */
    unsigned char * srcfilename;       /* source file name       */
    unsigned char * destfilename;      /* destination file name  */

    printf( "\n" );

    if( argc != 4 ) {
        printf( "Usage : aget <LU name> <source filename>"  \
            " <destination filename> \n");
        exit( -1 );
    }

    LU_name = argv[1];
    srcfilename = argv[2];
    destfilename = argv[3];
/* Create the connection object          */
    aftp_create ( connection_id, &aftp_rc );

    if ( aftp_rc != AFTP_RC_OK ) {
        printf ( "Error creating connection object.\n" );
```

```
    exit ( -1 );
}
```

Set the partner LU name as the destination.

```
aftp_set_destination (
    connection_id,
    (unsigned char AFTP_PTR)LU_name,
    (AFTP_LENGTH_TYPE)strlen ( LU_name ),
    &aftp_rc );

if( aftp_rc != AFTP_RC_OK ) {
    printf ( "Error setting the destination.\n" );
    exit ( -1 );
}
```

Set the security to NONE unless you need security.

```
aftp_set_security_type (
    connection_id,
    AFTP_SECURITY_NONE,
    &aftp_rc );

if ( aftp_rc == AFTP_RC_OK ) {
    printf ( "Setting security type to NONE.\n" );
} else {
    printf ( "Error setting security type.\n" );
}
```

Establish a connection with AFTPD server.

```
aftp_connect ( connection_id, &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {
    printf ( "Error establishing the connection.\n" );
    exit ( -1 );
}
```

Set up file transfer mode.

```
aftp_set_write_mode (
   connection_id,
   AFTP_REPLACE,
   &aftp_rc );

if ( aftp_rc != AFTP_RC_OK ) {
    printf ( "Error setting write mode.\n" );
}
```

Extract the security type and display it.

```
aftp_extract_security_type (
   connection_id,
   &sec_type,
   &aftp_rc );

if ( aftp_rc == AFTP_RC_OK ) {
    printf ( "Security type is : %lu\n", sec_type );
```

```
    } else {
        printf ( "Error extracting security type.\n" );
    }
```

Transfer the file from the server to the client.

```
    aftp_receive_file (
        connection_id,
        (unsigned char AFTP_PTR)destfilename,
        (AFTP_LENGTH_TYPE)strlen ( destfilename ),
        (unsigned char AFTP_PTR)srcfilename,
        (AFTP_LENGTH_TYPE)strlen ( srcfilename ),
        &aftp_rc );

    if ( aftp_rc == AFTP_RC_OK ) {
        printf ( "File successfully transfered.\n" );
    }
```

This is an example of how to show error reporting.

```
    AFTP_LENGTH_TYPE return_length;
    char error_string[ AFTP_MESSAGE_SIZE ];

    printf ( "Error %lu transfering the file.\n", aftp_rc );
```

Specify a detail level according to how much information you want returned. In this case, return code information is requested.

```
    aftp_format_error (
       connection_id,
       (AFTP_DETAIL_LEVEL_TYPE)AFTP_DETAIL_RC,
       (unsigned char AFTP_PTR)error_string,
       (AFTP_LENGTH_TYPE)( sizeof ( error_string )-1 ),
       &return_length,
       &aftp_rc );
```

Add a null terminator.

```
        error_string[ return_length ] = '\0';
        printf ( "%s", error_string );
    }
```

Close a connection with AFTPD server.

```
    aftp_close ( connection_id, &aftp_rc );

    if ( aftp_rc != AFTP_RC_OK ) {
        printf ( "Error closing the connection.\n" );
        exit ( -1 );
    }
```

Destroy the connection identifier.

```
    aftp_destroy ( connection_id, &aftp_rc );

    if (aftp_rc != AFTP_RC_OK) {
```

```
        printf ( "Error destroying the connection id.\n" );
        exit ( -1 );
    }

    return(0);
}

/* END SAMPLE PROGRAM */
```

Internationalization

This section describes the features available in Microsoft® Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0 for supporting international languages and different national language character sets. Introduced with SNA Server version 3.0, the SNANLS API available on Microsoft Windows® XP, Windows 2000, Windows NT®, Windows Millennium Edition, Windows 98, and Windows 95 can be used to support international languages in applications for Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0 by means of a standardized and consistent interface. The SNANLS API uses the language support features provided with Windows XP/2000/NT and extends these same features to Windows Millennium Edition, Windows 98, and Windows 95 clients. SNANLS supports European languages that use single-byte encoding as well as East Asia languages that use double-byte or Unicode encoding.

In Windows 3.x and MS-DOS® environments where SNANLS is not supported, developers can use the TrnsDT API for supporting East Asia languages such as Japanese, Korean, and Chinese, and Windows code page support for single-byte conversions.

This section contains:

SNA National Language Support

Microsoft® Host Integration Server 2000 is composed of many components, including 3270 Client, 5250 Client, NetView services, MSMQ-MQSeries bridge, COM Transaction Integrator, and OLE DB providers. In past versions of SNA Server, most components used a variety of methods for supporting different national languages and character sets and for converting between EBCDIC host character code sets and ANSI code pages for the PC. For East Asia languages such as Japanese, Korean, and Chinese, the TrnsDT API was used for double-byte character stream (DBCS) conversions. For other languages, a component's own proprietary functions were often used for single-byte character stream (SBCS) conversions. It was common for two components or applications to use different functions for converting strings from EBCDIC to ANSI and from ANSI to EBCDIC. The methods used were quite varied and the code page support was not entirely consistent across products.

All of this changed with the release of SNA Server version 3.0 which introduced a standard SNA National Language Support (SNANLS) API for supporting national languages. The SNANLS API was developed to standardize the way in which national languages and locales are supported on SNA Server. SNANLS was designed to handle string conversion necessary for supporting a wide range of host and PC code pages. The new components developed for SNA Server 3.0 and later, such as the Host Print Service and Shared Folders Service, use SNANLS API to convert strings from EBCDIC to ANSI and from ANSI to EBCDIC.

The SNA National Language Support API is the standard means to convert strings in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0. SNANLS presents a single interface to applications that need strings converted from one code page to another. These conversions may be EBCDIC-to-ANSI, ANSI-to-EBCDIC, EBCDIC-to OEM code pages, OEM-to-EBCDIC, EBCDIC-to-ISO code pages, and ISO-to-EBCDIC. Additionally, SNANLS supports the broadest possible range of host and PC code page conversions.

SNANLS provides a uniform interface for programmers, hiding the details and difficulties of string conversion. SNANLS supports both SBCS and DBCS conversions. The actual string conversion is handled by two other lower-level APIs. For SBCS conversions, SNANLS uses the system-provided Win32® NLS API that is resident on Microsoft Windows® XP, Windows 2000, Windows NT® 4.0, and Windows NT 3.51. For use on Windows Millennium Edition, Windows 98, and Windows 95 systems, Host Integration Server 2000 and SNA Server include a version of the SNANLS DLL which provides integrated support for NLS and Unicode conversions. For DBCS conversions, SNANLS uses the TrnsDT API developed by the SNA Server team. The TrnsDT API is installed with Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS is supported on Windows XP, Windows 2000, Windows NT, Windows Millennium Edition, Windows 98, and Windows 95. The goal for all future development is to use only SNANLS.

This section contains:

National Language Support in Windows 2000 and Windows NT

National Language Support (NLS) provides a standardized method of supporting multiple international locales, code pages, input methods, sort orders, and number/currency/time/date formats. The Win32 NLS API provides developers with a way to access system-provided Unicode-to-ANSI and ANSI-to-Unicode conversion services. Windows 2000 and Windows NT 4.0 are supplied with EBCDIC-to-Unicode and Unicode-to-EBCDIC translation tables for all of the popular host code pages.

The SNANLS API leverages the existing work done to support the NLS API on Windows 2000 and on Windows NT 3.51 and 4.0. Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0 take advantage of these EBCDIC-to-Unicode-to-ANSI and ANSI-to-Unicode-to-EBCDIC code page conversion services. Currently, the Win32 NLS API only supports SBCS EBCDIC code pages. However, future versions of the NLS API will support DBCS EBCDIC. SNANLS currently uses TrnsDT for DBCS conversions.

SNANLS Code Page Support

The SNANLS API provides functions for converting single-byte character stream (SBCS) EBCDIC-to-Unicode-to-ANSI and SBCS ANSI-to-Unicode-to-EBCDIC by leveraging the Win32 National Language Support (NLS) API. The Win32 NLS API uses resource files containing NLS conversion tables that are installed on the target PC when Windows is installed or by the setup program for Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0 (the setup program also adds the required registry entries). The SNANLS DLL is supplied with Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS supports conversions for the following groups of code pages:

- ANSI code pages
- ANSI/OEM code pages
- EBCDIC code pages
- OEM PC code pages
- Open Systems code pages
- ISO code pages

The following tables list the code page support by category provided using SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

This section contains:

ANSI Code Page Support Using SNANLS

The following table shows the ANSI code pages and character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	HOST CCSID	Type	NLS Filename	Comments
ANSI - Arabic	1256	1256	SBCS	c_1256.nls	
ANSI - Baltic	1257	1257	SBCS	c_1257.nls	
ANSI - Cyrillic	1251	1251	SBCS	c_1251.nls	
ANSI - Central Europe	1250	1250	SBCS	c_1250.nls	
ANSI - Greek	1253	1253	SBCS	c_1253.nls	
ANSI - Hebrew	1255	1255	SBCS	c_1258.nls	
ANSI - Latin I	1252	1252	SBCS	c_1252.nls	
ANSI - Turkish	1254	1254	SBCS	c_1254.nls	

Note that all of these ANSI code pages support the Euro.

ANSI/OEM Code Page Support Using SNANLS

The following table shows the ANSI/OEM code pages and character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	Host CCSID	Type	NLS Filename	Comments
ANSI/OEM - Japanese Shift-JIS	932	932	SBCS	c_932.nls	Japanese JIS-8 Bit + Shift-JIS
ANSI/OEM - Korean	949	949	DBCS	c_949.nls	Korean Hangul (Extended Wansung)
ANSI/OEM - Simplified Chinese GBK	936	936	DBCS	c_936.nls	Simplified Chinese GBK
ANSI/OEM - Thai	874	874	SBCS	c_874.nls	Thai
ANSI/OEM - Traditional Chinese Big5	950	950	DBCS	c_950.nls	Traditional Chinese Big5
ANSI/OEM - Viet Nam	1258	1258	SBCS	c_1258.nls	Viet Nam

Note that none of these code pages support the Euro.

EBCDIC Code Page Support Using SNANLS

The following table shows the EBCDIC code pages and character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	Host CCSID	Euro	Supported by SNANLS in SNA Server 3.0 and later
EBCDIC - Arabic	20420	420		partial (See Note)
EBCDIC - Cyrillic (Russian)	20880	880		yes
EBCDIC - Cyrillic (Serbian, Bulgarian)	21025	1025		yes
EBCDIC - Denmark/ Norway (Euro)	1142	277	yes	yes
EBCDIC - Denmark/ Norway	20277	277		yes
EBCDIC - Finland/ Sweden (Euro)	1143	278	yes	yes
EBCDIC - Finland/ Sweden	20278	278		yes
EBCDIC - France (Euro)	1147	297	yes	yes
EBCDIC - France	20297	297		yes
EBCDIC - Germany (Euro)	1141	273	yes	yes
EBCDIC - Germany	20273	273		yes
EBCDIC - Greek (Modern)	875	875		yes
EBCDIC - Greek	20423	423		yes
EBCDIC - Hebrew	20424	424		partial (See Note)
EBCDIC - Icelandic (Euro)	1149	871	yes	yes
EBCDIC - Icelandic	20871	871		yes
EBCDIC - International (Euro)	1148	500	yes	yes
EBCDIC - International	500	500		yes
EBCDIC - Italy (Euro)	1144	280	yes	yes
EBCDIC - Italy	20280	280		yes
EBCDIC - Japan English (Extended)	1027			
EBCDIC - Japan English/Kanji (Extended)	939	939		yes
EBCDIC - Japan English/Kanji (Extended)	5035			
EBCDIC - Japan Katakana (Extended)	290	290		yes
EBCDIC - Japan Katakana/Kanji (Extend Katakana)	930	930		yes
EBCDIC - Japan Katakana/Kanji (Extend Katakana)	5026			
EBCDIC - Japanese	931	931		yes
EBCDIC - Korea (Extended)	933	933		yes
EBCDIC - Latin America/ Spain (Euro)	1145	284	yes	yes
EBCDIC - Latin America/ Spain	20284	284		yes
EBCDIC - Multilingual/ ROECE (Latin-2)	870	870		yes
EBCDIC - Simplified Chinese (Extended)	935	935		yes
EBCDIC - Thai	20838	838		yes
EBCDIC - Traditional Chinese (Extended)	937	937		yes
EBCDIC - Turkish (Latin-3)	20905	905		yes
EBCDIC - Turkish (Latin-5)	1026	1026		yes

EBCDIC - U.S./ Canada (Euro)	1140	37	yes	yes
EBCDIC - U.S./ Canada	37	37		yes
EBCDIC - United Kingdom (Euro)	1146	285	yes	yes
EBCDIC - United Kingdom	20285	285		yes

None of the code pages supporting the Euro are supplied or installed with SNA Server 3.0

Support for Arabic and Hebrew code page conversions are limited to left-to-right output. Bidirectional output including the default Arabic and Hebrew right-to-left output is not supported in this release of Host Integration Server 2000.

ISO Code Page Support Using SNANLS

The following table shows the ISO NLS code pages and host character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	Host CCSID	Euro	Supported by SNANLS in SNA Server 3.0 and later
ISO 6937 Non-Spacing Accent	20269	6937		
ISO 8859-1 Latin-1	28591	819		
ISO 8859-15 Latin 9 (Euro)	20865	923	yes	
ISO 8859-2 Central Europe	28592	912		
ISO 8859-3 Latin 3	28593	913		
ISO 8859-4 Baltic	28594	914		
ISO 8859-5 Cyrillic	28595	915		
ISO 8859-6 Arabic	28596	1089		
ISO 8859-7 Greek	28597	813		
ISO 8859-8 Hebrew (Visually Ordered)	28598	916		
ISO 8859-9 Hebrew (Logically Ordered)	28599	920		

OEM PC Code Page Support Using SNANLS

The following table shows the OEM PC code pages and character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	Host CCSID	Type	NLS Filename	Comments
OEM - Arabic	864	864	SBCS	c_864.nls	
OEM - Baltic	775	775	SBCS	c_775.nls	
OEM - Canadian French	863	863	SBCS	c_863.nls	OEM - Canada (850 subset)
OEM - Cyrillic	855	855	SBCS	c_855.nls	
OEM - Cyrillic II	866	866	SBCS	c_866.nls	OEM - Russian
OEM - Greek 437G	737	737	SBCS	c_737.nls	
OEM - Hebrew	862	862	SBCS	c_862.nls	
OEM - Icelandic	861	861	SBCS	c_861.nls	OEM - Iceland
OEM - Modern Greek	869	869	SBCS	c_869.nls	
OEM - Multilingual Latin I	850	850	SBCS	c_850.nls	
OEM - Multilingual Latin II	852	852	SBCS	c_852.nls	
OEM - Nordic	865	865	SBCS	c_865.nls	OEM - Denmark, Norway, Finland, Sweden
OEM - Portuguese	860	860	SBCS	c_860.nls	OEM - Portugal (850 subset)
OEM - Turkish	857	857	SBCS	c_857.nls	
OEM - United States	437	437	SBCS	c_437.nls	

Note that none of these code pages support the Euro.

Open Systems Code Page Support Using SNANLS

The following table shows the Open Systems NLS code pages and host character code set identifiers (CCSIDs) supported by SNANLS in Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0.

SNANLS Display Name	NLS Code Page	Host CCSID	Euro	Supported by SNANLS in SNA Server 3.0 and later
Latin-1/Open System (Euro)	20924	924	yes	
Latin-1/Open System	1047	1047		

SNANLS Dependencies

The only file required to support SNANLS API on Windows XP, Windows 2000, and Windows NT is SNANLS.DLL. To link to this DLL use the SNANLS.H header (located under the \SDK\INCLUDE subdirectory) and the SNANLS.LIB library file (located under the \SDK\LIB subdirectory) supplied with the Host Integration Server 2000 SDK. The SNANLS.H include file and the SNANLS.LIB file are also provided as part of the Microsoft Developer Network (MSDN) Platform SDK and with Visual Studio 6.0 or later. Note that individual Win32 NLS resource files must be installed in order to support the various languages and code pages on Windows XP, Windows 2000, Windows NT, Windows Millennium Edition, Windows 98, and Windows 95.

The Win32 NLS files needed to support various languages are normally installed when the operating system is installed during setup for Windows XP or Windows 2000. On Windows NT 3.51 or 4.0, the individual Win32 NLS files needed to support various languages are not normally installed during setup. The EBCDIC Win32 NLS files were delivered as part of the Windows NT 4.0 Language Pack (see the \LANGPACK subdirectory on the Windows NT 4.0 CD-ROM). Copies of these Win32 NLS support files are bundled with Host Integration Server 2000. The Win32 NLS resource files for EBCDIC code pages are installed on the server machine and client computers by the setup program for Host Integration Server 2000. Win32 NLS files for EBCDIC code pages are also bundled with SNA Server 4.0 and SNA Server 3.0, but the list of code pages is a subset of those supplied with Host Integration Server 2000. The Win32 NLS resource files are installed on the server machine and client computers by the setup program for SNA Server 4.0. The required Win32 NLS resource files are installed on only server machine by the setup program for SNA Server 3.0 when the Print Service option is selected. No Win32 NLS resource files are installed on the client computers by setup for SNA Server 3.0.

For SNA Server 3.0 client applications that use SNA NLS on Windows NT, the necessary NLS files may need to be copied from the SNA Server CD-ROM to the client machine. For Windows NT clients, the NLS files must be copied to the Windows NT system directory (default location is C:\WINNT\SYSTEM32) from the \SDK\SNANLS\WINNT directory on the SNA Server CD-ROM. Windows NT 4.0 includes a number of NLS files for some ANSI, ISO, and OEM code pages that are copied to the system directory (default location is C:\WINNT\SYSTEM32) when the operating system is installed. But if additional NLS files (EBCDIC code pages, for example) are required, these must be copied from the SNA Server CD-ROM.

For SNA Server 3.0 client applications that use SNA NLS on Windows Millennium Edition, Windows 98, and Windows 95, the necessary NLS files will need to be copied from the SNA Server CD-ROM to the client machine. For SNA clients running on Windows Millennium Edition, Windows 98, and Windows 95, the NLS files must be copied to the \WINDOWS\SYSTEM subdirectory from the \SDK\SNANLS\WIN9X directory on the SNA Server CD-ROM. Windows Millennium Edition and Windows 98 include a number of NLS files for some ANSI, ISO, and OEM code pages that are copied to the Windows system directory (default location is C:\WINDOW\SYSTEM) when the operating system is installed. But if additional NLS files (EBCDIC code pages, for example) are required, these must be copied from the SNA Server CD-ROM.

The registry settings required to use specific NLS files are enabled on Windows XP and Windows 2000 when the operating system is installed. When the End-user client or Administrator clients from Host Integration Server 2000 are installed, the registry settings required to use specific NLS files are automatically created. On Windows NT server computer, the registry settings required to use specific NLS files are enabled by the setup program for Host Integration Server 2000 or by the setup program for SNA Server 4.0 or SNA Server 3.0.

For SNA Server 3.0 clients on Windows NT, Windows Millennium Edition, Windows 98, and Windows 95, the appropriate registry settings will need to be set manually or by a vendor-supplied client installation program for EBCDIC code pages and other NLS code pages not installed by the operating system. The registry settings required for common EBCDIC code pages are shown in the table below.

File name	SNANLS Display Name	NLS Code Page	Host CCSID	Registry setting
c_037.nls	EBCDIC - U.S./ Canada	37	37	Value Name=37 Type=REG_SZ Data=c_037.nls
c_500.nls	EBCDIC - International	500	500	Value Name=500 Type=REG_SZ Data=c_500.nls
c_870.nls	EBCDIC - Multilingual/ ROECE (Latin-2)	870	870	Value Name=870 Type=REG_SZ Data=c_870.nls
c_875.nls	EBCDIC - Greek (Modern)	875	875	Value Name=875 Type=REG_SZ Data=c_875.nls

c_1026.nls	EBCDIC - Turkish (Latin-5)	1026	1026	Value Name=1026 Type=REG_SZ Data=c_1026.nls
c_20273.nls	EBCDIC - Germany	20273	273	Value Name=20273 Type=REG_SZ Data=c_20273.nls
c_20277.nls	EBCDIC - Denmark/ Norway	20277	277	Value Name=20277 Type=REG_SZ Data=c_20277.nls
c_20278.nls	EBCDIC - Finland/ Sweden	20278	278	Value Name=20278 Type=REG_SZ Data=c_20278.nls
c_20280.nls	EBCDIC - Italy	20280	280	Value Name=20280 Type=REG_SZ Data=c_20280.nls
c_20284.nls	EBCDIC - Latin America/ Spain	20285	284	Value Name=20284 Type=REG_SZ Data=c_20284.nls
c_20285.nls	EBCDIC - United Kingdom	20285	285	Value Name=20285 Type=REG_SZ Data=c_20285.nls
c_20297.nls	EBCDIC - France	20297	297	Value Name=20297 Type=REG_SZ Data=c_20297.nls
c_20420.nls	EBCDIC - Arabic	20420	420	Value Name=28596 Type=REG_SZ Data=c_20420.nls
c_20423.nls	EBCDIC - Greek	20423	423	Value Name=20423 Type=REG_SZ Data=c_20423.nls
c_20424.nls	EBCDIC - Hebrew	20424	424	Value Name=20424 Type=REG_SZ Data=c_20424.nls
c_20838.nls	EBCDIC - Thai	20838	838	Value Name=20838 Type=REG_SZ Data=c_20838.nls
c_20871.nls	EBCDIC - Icelandic	20871	871	Value Name=20871 Type=REG_SZ Data=c_20871.nls
c_20880.nls	EBCDIC - Cyrillic (Russian)	20880	880	Value Name=20880 Type=REG_SZ Data=c_20880.nls
c_20905.nls	EBCDIC - Turkish (Latin-3)	20905	905	Value Name=20905 Type=REG_SZ Data=c_20905.nls
c_21025.nls	EBCDIC - Cyrillic (Serbian, Bulgarian)	21025	1025	Value Name=21025 Type=REG_SZ Data=c_21025.nls

Note On Windows XP, Windows 2000, Windows NT, Windows Millennium Edition, Windows 98, and Windows 95, the registry settings are located under the **HKEY_LOCAL_MACHINE** under the following subkey: SYSTEM\CurrentControlSet\Control\Nls\CodePage

Windows Millennium Edition, Windows 98, and Windows 95 do not natively support Unicode, which is necessary to support the Win32 NLS API. Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0 include the necessary support for Unicode in the SNANLS.DLL file supplied for these platforms. Since Windows Millennium Edition, Windows 98, and Windows 95 do not come with the Win32 NLS files needed to support various languages (EBCDIC code pages, for example), these NLS files are supplied with Host Integration Server 2000, SNA Server 4.0, and SNA Server 3.0. Host Integration Server 2000 and SNA Server 4.0 automatically install these files on client computers. For use with SNA Server 3.0 clients, these files are located in the \SDK\SNANLS\WIN9X subdirectory on the SNA Server CD-ROM. Note that because of system differences, the NLS files for

Windows XP, Windows 2000, and Windows NT and the NLS files for Windows Millennium Edition, Windows 98, and Windows 95 are different and may not be interchanged.

SNANLS API Functions

The SNANLS API is documented in the SNANLS.H file in the software development kit (SDK) provided with Microsoft® Host Integration Server 2000 and SNA Server 3.0 and SNA Server 4.0.

The following functions are supported by SNANLS on Host Integration Server 2000:

- CloseNlsRegistry
- FindCloseCodePage
- FindFirstCodePage
- FindNextCodePage
- GetCodePage
- GetCodePageDisplayStr
- IsInstalledCodePage
- OpenNlsRegistry
- SnaNlsInit
- SnaNlsMapString

The following functions are supported by SNANLS on SNA Server 4.0 and SNA Server 3.0:

- CloseNlsRegistry
- IsInstalledCodePage
- OpenNlsRegistry
- SnaNlsInit
- SnaNlsMapString

CloseNlsRegistry

The SNANLS **CloseNlsRegistry** function closes an open registry key on a local or remote computer.

```
BOOL WINAPI CloseNlsRegistry(
    HKEY KeyHandle
);
```

Parameters

KeyHandle
Supplied parameter. The handle to a key in the registry opened using **OpenNlsRegistry**.

Return Values

The **CloseNlsRegistry** function returns zero on success, otherwise a non-zero value is returned on failure.

Remarks

The *KeyHandle* parameter passed to this function is the handle returned from a previous call to the **OpenNlsRegistry** function. This function is primarily used by the Print Service in Host Integration Server 2000 and SNA Server 3.0 and later to determine what code pages are supported on a remote computer providing the print services function.

This function is supported by SNANLS on Host Integration Server 2000 and SNA Server 3.0 and later.

FindCloseCodePage

The SNANLS **FindCloseCodePage** function closes the handle allocated by a call to the **FindFirstCodePage** function.

```
BOOL WINAPI FindCloseCodePage(
    const HANDLE hInfo
);
```

Parameters

hInfo
Supplied parameter. The handle allocated and returned using **FindFirstCodePage**.

Return Values

The **FindCloseCodePage** function returns TRUE on success, otherwise the returned value on failure is FALSE.

Remarks

The *hInfo* parameter passed to this function is the handle returned from a previous call to the **FindFirstCodePage** function.

This function is supported by SNANLS on Host Integration Server 2000.

FindFirstCodePage

The SNANLS **FindFirstCodePage** function finds the first instance of a code page satisfying the condition specified, copies the code page information to a structure passed as a parameter, opens and returns a handle used in subsequent calls to the **FindNextCodePage** function.

```
const HANDLE WINAPI FindFirstCodePage(
    DWORD dwEnumOption,
    struct CodePage *pPage
);
```

Parameters

dwEnumOption

Supplied parameter. The set of conditions that a code page should satisfy. These conditions can be any combination of the following values defined in the SNANLS.h include file:

ENUM_CP_AVAILABLE (0x01)

The code page is installed and available for use.

ENUM_CP_HOST (0x02)

The code page is a host code page (EBCDIC, for example).

ENUM_CP_EURO (0x04)

The code page contains support for the Euro character.

ENUM_CP_DBCS (0x08)

The code page is for a double-byte character set.

ENUM_CP_MBCS (0x10)

The code page is for a mixed-byte character set.

ENUM_CP_SBCS (0x20)

The code page is for a single-byte character set.

Note that some of these combinations represent cases that will not match any installed code pages used by SNANLS.

pPage

Supplied and returned parameter. A pointer to a struct CodePage where the code page information should be copied.

On a successful return, the memory location pointed to by this parameter will be filled with the information for the first code page satisfying the conditions in *dwEnumOption*. On failure, no changes will be made to the memory pointed to by this parameter.

The CodePage struct is defined in the SNANLS.H include file as follows:

```
struct CodePage {
    BYTE    CodePageKey;
    DWORD   CodePageID;
    WCHAR   szFriendlyName[CP_SIZE];
    short   eGroup;
    BOOL    bAvailable;
    BYTE    bccsid;
    BOOL    bEuro;
};
```

The members of this CodePage structure are as follows:

CodePageKey

A numeric value that represents the index into the array of CodePage structures. This value should be used as an opaque value, since this value can be changed arbitrarily by Service Packs when additional code pages are supported.

CodePageID

The NLS code page number.

szFriendlyName

The SNANLS display name for this code page.

eGroup

The group that this code page is represented by. .This value can be one of the following enumerations defined in the SNANLS.h include file for code groups:

ENUM_CP_EBCDIC

This code page is a member of the EBCDIC code page group.

ENUM_CP_ANSI

This code page is a member of the ANSI code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEMPC

This code page is a member of the OEM PC code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEM PC

This code page is a member of the OEM PC code page group.

ENUM_CP_OPEN

This code page is a member of the Open Systems code page group.

ENUM_CP_UCS

This code page is a member of the UCS code page group.

bAvailable

A boolean used to indicate that this code page is installed on the computer. A value of FALSE for this member indicates that the computer will not be queried to determine if this code page is installed. A value of TRUE indicated the code page is installed.

bccsid

A flag used to indicate the type of code page. This flag can be one of the following:

ENUM_CP_DBCS (0x08)

The code page is for a double-byte character set.

ENUM_CP_MBCS (0x10)

The code page is for a mixed-byte character set.

ENUM_CP_SBCS (0x20)

The code page is for a single-byte character set.

bEuro

A boolean value used to indicate if this code page supports the Euro symbol. If this value is TRUE, then the Euro symbol is supported.

Return Values

The **FindFirstCodePage** function returns a handle used in calls to the **FindNextCodePage** or **FindCloseCodePage** on success.

On failure, INVALID_HANDLE_VALUE is returned for the value of this handle.

Remarks

The handle returned by this function should not be tampered with by the user.

This function is supported by SNANLS on Host Integration Server 2000.

FindNextCodePage

The SNANLS **FindNextCodePage** function finds the next instance of code page satisfying the condition specified in the initial call to the **FindFirstCodePage** function, and copies the code page information to a structure passed as a parameter.

```
BOOL WINAPI FindNextCodePage(
    const HANDLE hInfo
    struct CodePage *pPage
);
```

Parameters

hInfo
Supplied parameter. The handle allocated and returned using **FindFirstCodePage**.

pPage
Supplied and returned parameter. A pointer to struct CodePage where the code page information should be copied.

On a successful return, the memory location pointed to by this parameter will be filled with the information for the next code page satisfying the conditions in *dwEnumOption* parameter passed to the FindFirstCodePage function.

On failure, no changes will be made to the memory pointed to by this parameter.

The CodePage struct is defined in the SNANLS.H include file as follows:

```
struct CodePage {
    BYTE    CodePageKey;
    DWORD   CodePageID;
    WCHAR   szFriendlyName[CP_SIZE];
    short   eGroup;
    BOOL    bAvailable;
    BYTE    bccsid;
    BOOL    bEuro;
};
```

The members of this CodePage structure are as follows:

CodePageKey

A numeric value that represents the index into the array of CodePage structures. This value should be used as an opaque value, since this value can be changed arbitrarily by Service Packs when additional code pages are supported.

CodePageID

The NLS code page number.

szFriendlyName

The SNANLS display name for this code page. The character string is null terminated.

eGroup

The group that this code page is represented by. .This value can be one of the following enumerations defined in the SNANLS.h include file for code groups:

ENUM_CP_EBCDIC

This code page is a member of the EBCDIC code page group.

ENUM_CP_ANSI

This code page is a member of the ANSI code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEMPC

This code page is a member of the OEM PC code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEM PC

This code page is a member of the OEM PC code page group.

ENUM_CP_OPEN

This code page is a member of the Open Systems code page group.

ENUM_CP_UCS

This code page is a member of the UCS code page group.

bAvailable

A boolean used to indicate that this code page is installed on the computer. A value of FALSE for this member indicates that the computer will not be queried to determine if this code page is installed. A value of TRUE indicated the code page is installed.

bccsid

A flag used to indicate the type of code page. This flag can be one of the following:

ENUM_CP_DBCS (0x08)

The code page is for a double-byte character set.

ENUM_CP_MBCS (0x10)

The code page is for a mixed-byte character set.

ENUM_CP_SBCS (0x20)

The code page is for a single-byte character set.

bEuro

A boolean value used to indicate if this code page supports the Euro symbol. If this value is TRUE, then the Euro symbol is supported.

Return Values

The **FindNextCodePage** function returns a value of TRUE on success. On failure, the returned value is FALSE.

Remarks

This function is supported by SNANLS on Host Integration Server 2000.

GetCodePage

The SNANLS **GetCodePage** function copies the code page information identified by a key to a structure passed as a parameter.

```
BOOL WINAPI GetCodePage(
    int nKey
    struct CodePage *pPage
);
```

Parameters

nKey

Supplied parameter. The numeric key to a code page. This value is an opaque index into an array containing the code pages supported by SNANLS. This value is normally the *CodePageKey* member of a CodePage structure returned from a previous call to **FindFirstCodePage** or **FindNextCodePage**.

pPage

Supplied and returned parameter. A pointer to struct CodePage where the code page information should be copied.

On a successful return, the memory location pointed to by this parameter will be filled with the information for the specific code page.

On failure, no changes will be made to the memory pointed to by this parameter.

The CodePage struct is defined in the SNANLS.H include file as follows:

```
struct CodePage {
    BYTE    CodePageKey;
    DWORD   CodePageID;
    WCHAR   szFriendlyName[CP_SIZE];
    short   eGroup;
    BOOL    bAvailable;
    BYTE    bccsid;
    BOOL    bEuro;
};
```

The members of this CodePage structure are as follows:

CodePageKey

A numeric value that represents the index into the array of CodePage structures. This value should be used as an opaque value, since this value can be changed arbitrarily by Service Packs when additional code pages are supported.

CodePageID

The NLS code page number.

szFriendlyName

The SNANLS display name for this code page. The character string is null terminated.

eGroup

The group that this code page is represented by. .This value can be one of the following enumerations defined in the SNANLS.h include file for code groups:

ENUM_CP_EBCDIC

This code page is a member of the EBCDIC code page group.

ENUM_CP_ANSI

This code page is a member of the ANSI code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEMPC

This code page is a member of the OEM PC code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_ISO

This code page is a member of the ISO code page group.

ENUM_CP_OEM PC

This code page is a member of the OEM PC code page group.

ENUM_CP_OPEN

This code page is a member of the Open Systems code page group.

ENUM_CP_UCS

This code page is a member of the UCS code page group.

bAvailable

A boolean used to indicate that this code page is installed on the computer. A value of FALSE for this member indicates that the computer will not be queried to determine if this code page is installed. A value of TRUE indicated the code page is installed.

bccsid

A flag used to indicate the type of code page. This flag can be one of the following:

ENUM_CP_DBCS (0x08)

The code page is for a double-byte character set.

ENUM_CP_MBCS (0x10)

The code page is for a mixed-byte character set.

ENUM_CP_SBCS (0x20)

The code page is for a single-byte character set.

bEuro

A boolean value used to indicate if this code page supports the Euro symbol. If this value is TRUE, then the Euro symbol is supported.

Return Values

The **GetCodePage** function returns a value of TRUE on success. On failure, the returned value is FALSE.

Remarks

This function is supported by SNANLS on Host Integration Server 2000.

GetCodePageDisplayStr

The SNANLS **GetCodePageDisplayStr** function copies the SNANLS code page display name identified by a key to a character string passed as a parameter.

```
BOOL WINAPI GetCodePageDisplayStr(
    int nKey
    WCHAR *szDisplayStr
);
```

Parameters

nKey
Supplied parameter. The numeric key to a code page. This value is an opaque index into an array containing the code pages supported by SNANLS. This value is normally the *CodePageKey* member of a CodePage structure returned from a previous call to **FindFirstCodePage** or **FindNextCodePage**.

szDisplayStr
Supplied and returned parameter. A pointer to a wide-character array where the SNANLS display name for the specific code page should be copied.

On a successful return, the memory location pointed to by this parameter will be filled with the SNANLS display name for the specific code page. The character string is null terminated.

On failure, no changes will be made to the memory pointed to by this parameter.

Return Values

The **GetCodePageDisplayStr** function returns a value of TRUE on success. On failure, the returned value is FALSE.

Remarks

This function is supported by SNANLS on Host Integration Server 2000.

IsInstalledCodePage

The SNANLS **IsInstalledCodePage** function determines if a code page is installed on a local or remote computer.

```
BOOL WINAPI IsInstalledCodePage(
    UINT CodePage,
    HKEY KeyHandle
);
```

Parameters

CodePage
Supplied parameter. The NLS code page.
KeyHandle
Supplied parameter. The registry key returned from a call to the **OpenNlsRegistry** function.

Return Values

The **IsInstalledCodePage** function returns non-zero if a code page is installed, otherwise a zero value is returned on failure.

Remarks

This function is primarily used by the Print Service in Host Integration Server 2000 and SNA Server 3.0 and later to determine if what code pages are supported on a remote computer providing the print services function.

This function is supported by SNANLS on Host Integration Server 2000 and SNA Server 3.0 and later.

OpenNlsRegistry

The SNANLS **OpenNlsRegistry** function opens a registry key on a local or remote computer pointing to the NLS Codepage registry entries.

```
HKEY WINAPI OpenNlsRegistry(
    char *MachineName,
    HKEY hkey,
    LPSTR Path
);
```

Parameters

MachineName
Supplied parameter. The name of the remote computer on which to open the registry. If this parameter is NULL, the registry on the local computer is opened.

hKey
Supplied parameter. The key to the registry to open. If this parameter is NULL, the HKEY_LOCAL_MACHINE key is used.

Path
Supplied parameter. The path to the key value in the registry hive to open. If this parameter is NULL, the following key is opened:

SYSTEM\CurrentControlSet\NLS\CodePage.

Return Values

The **OpenNlsRegistry** function returns a handle to the opened registry key on success, otherwise a NULL value is returned on failure.

Remarks

This function is primarily used by the Print Service in Host Integration Server 2000 and SNA Server 3.0 and later to determine if what code pages are supported on a remote computer providing the print services function.

This function is supported by SNANLS on Host Integration Server 2000 and SNA Server 3.0 and later.

SnaNlsInit

The **SnaNlsInit** function is called to determine if the code page needed is supported by code page translations using SNANLS. This allows an application to determine if the necessary NLS language files containing code page translation tables are installed on the local system.

```
int WINAPI SnaNlsInit(
    UINT CodePage
);
```

Parameters

CodePage
Supplied parameter. The number of the NLS code page for which support is requested. The *CodePage* parameter corresponds with the registry settings on Windows 2000/NT located under the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Nls\CodePage** subkey.

Return Values

The **SnaNlsInit** function returns non-zero if code page translations are supported; otherwise 0 is returned.

Remarks

If CP_ACP (the current ANSI code page) is passed as the *CodePage* parameter, this functions returns non-zero.

This function is supported by SNANLS on Host Integration Server 2000 and SNA Server 3.0 and later.

SnaNlsMapString

The **SnaNlsMapString** function is called to translate a string from one code page to another.

```
int WINAPI SnaNlsMapString(
    LPCTSTR lpSrcStr,
    LPTSTR lpDestStr,
    UINT inCodePage,
    UINT outCodePage,
    int in_length,
    int out_length,
    UINT in_type,
    UINT out_type,
    WORD *Options
);
```

Parameters

lpSrcStr
Supplied parameter. The input source string to be translated.

lpDestStr
Returned parameter. The translated string which may be NULL if *out_length* was zero.

inCodePage
Supplied parameter. Specifies the code page of the incoming source string; ignored if the input is Unicode.

outCodePage
Supplied parameter. Specifies the code page of the output translated string; ignored if the output is Unicode.

in_length
Supplied parameter. Specifies the length of the input source string in characters if the input is multibyte or in wide characters if the input is Unicode.

out_length
Supplied parameter. Specifies the maximum length available for the output translated string in characters if the output is multibyte or in wide characters if the output is Unicode.

in_type
Supplied parameter. Specifies the type of the input source string. Possible values for *in_type* are SNA_MULTIBYTE for multibyte and SNA_UNICODE for Unicode.

out_type
Supplied parameter. Specifies the type of the output translated string. Possible values for *out_type* are SNA_MULTIBYTE for multibyte and SNA_UNICODE for Unicode.

Options
Supplied and returned parameter. As a supplied parameter, this specifies a set of options that may be applied to the translation process, including TrnsDT options and the default character for the translation. On return, this parameter indicates the required buffer length for the output translated string if the function call failed.

Return Values

The **SnaNlsMapString** function returns the number of characters or wide characters written to *lpDestStr* on success; otherwise 0 is returned on failure.

On failure, the Win32® GetLastError function should be used to return an error code indicating the cause of the failure. Possible values returned by GetLastError are as follows:

ERROR_NOT_SUPPORTED

This error is returned for two possible reasons—either the NLS language resource file is not available or the *in_type* and *out_type* of the source and destination strings are not of the same type.

ERROR_BUFFER_OVERFLOW

This error is returned if the output buffer is too small. In such cases, the *Options* parameter returns with the value needed for *out_length*.

ERROR_INVALID_PARAMETER

This error is returned if a bad value was passed in a parameter; for example, if the *in_type* or *out_type* parameters contained undefined values.

ERROR_INVALID_DATA

This error is returned if a bad value was passed in the *lpSrcStr* parameter; for example, if the input string has a lead byte at the end.

ERROR_OUTOFMEMORY

This error is returned if memory could not be allocated for use by the SNANLS DLL.

The TrnsDT API

The SNANLS API also allows applications to convert double-byte character stream (DBCS) EBCDIC-to-ANSI and DBCS ANSI-to-EBCDIC by leveraging another Host Integration Server 2000 API called TrnsDT. The TrnsDT API has its own mechanism to translate East Asia code pages using conversion table resource files (*.TBL files) that the setup program for Microsoft® SNA Server version 3.0 and later installs on the target PC.

In Microsoft Windows® version 3.x and Microsoft MS-DOS® environments where SNANLS is not supported, developers can use the TrnsDT API for supporting East Asia languages such as Japanese, Korean, and Chinese, and Windows code page support for single-byte conversions.

This section contains:

TrnsDT Code Page Support

The TrnsDT API is used to perform all DBCS EBCDIC-to- ASCII conversions throughout Host Integration Server 2000. To a degree, TrnsDT has been and continues to be a uniform translation method and cross-component resource. TrnsDT also handles mixed DBCS and SBCS, plus SBCS for Japan.

This section contains:

Host EBCDIC SBCS Using TrnsDT

The following table shows the character code set identifiers (CCSIDs) for EBCDIC single byte character sets (SBCS) supported by TrnsDT in Host Integration Server 2000 and in SNA Server 3.0 and later.

Code Page Display Name	Type	CCSID	Character Set	Comments
IBM EBCDIC - U.S./Canada	SBCS	037	697	IBM English lowercase
IBM EBCDIC - Japan Katakana (Extended)	SBCS	290	1172	IBM Extended English Katakana
IBM EBCDIC - Korean (Extended)	SBCS	833	934	Korean (Extended)
IBM EBCDIC - Simplified Chinese (Extended)	SBCS	836	935	Simplified Chinese single-byte
IBM EBCDIC - Japan English (Extended)	SBCS	1027	1172	IBM Extended lowercase English
IBM EBCDIC - Traditional Chinese (Extended)	SBCS	28709	937	Traditional Chinese (Extended)

Host EBCDIC DBCS Using TrnsDT

The following table shows the character code set identifiers (CCSIDs) for EBCDIC double byte character sets (DBCS) supported by TrnsDT in Host Integration Server 2000 and in SNA Server 3.0 and later.

Code Page Display Name	Type	CCSID	Character Set	Comments
IBM EBCDIC - Japan	DBCS	300	1001	IBM Japanese (including 4370 user-defined characters).
IBM EBCDIC - Korea	DBCS	834	933	IBM Korean (including 1880 user-defined characters).
IBM EBCDIC - Traditional Chinese	DBCS	835	937	Traditional Chinese Host double-byte (including 6204 user-defined characters)
IBM EBCDIC - Simplified Chinese	DBCS	837	837	Simplified Chinese Host double-byte
IBM EBCDIC - Japan	DBCS	4396	930, 931, 939	IBM Japanese (including 1880 user-defined characters).

Host EBCDIC Mixed SBCS and DBCS Using TrnsDT

The following table shows the character code set identifiers (CCSIDs) for EBCDIC mixed single byte character sets (SBCS) and double byte character sets (DBCS) supported by TrnsDT in Host Integration Server 2000 and in SNA Server 3.0 and later.

Code Page Display Name	**Type**	**CCSID**	**Comments**
IBM EBCDIC - Japan Katakana/Kanji (Extended)	Mixed	930	Japanese Katakana-Kanji mixed with 4370 user-defined characters.
IBM EBCDIC - Japanese	Mixed	931	Japan (English Lower-Case & Japanese)
IBM EBCDIC - Korea (Extended)	Mixed	933	Korean Mixed with 1880 user-defined characters.
IBM EBCDIC - Simplified Chinese (Extended)	Mixed	935	Simplified Chinese Host mixed with 1880 user-defined characters.
IBM EBCDIC - Traditional Chinese (Extended)	Mixed	937	Traditional Chinese Host mixed with 4370/6204 user-defined characters.
IBM EBCDIC - Japan English/Kanji (Extended)	Mixed	939	Japanese Latin Kanji mixed with 4370 user-defined characters.
IBM EBCDIC - Japan Katakana/Kanji (Extended)	Mixed	5026	A subset of CCSID 930 Japanese Katakana-Kanji mixed.
IBM EBCDIC - Japan English/Kanji (Extended)	Mixed	5035	A subset of CCSID 939 Japanese Latin Kanji mixed.

TrnsDT Conversions Possible

The following table describes conversions possible for TrnsDT.

Country/Region	Conversion	From CCSID	To CCSID
Japan	Host-PC	930	932
Japan	Host-PC	931	932
Japan	Host-PC	939	932
Japan	Host-PC	290	932
Japan	Host-PC	1027	932
Japan	Host-PC	5026	932
Japan	Host-PC	5035	932
Japan	PC-Host	932	930
Japan	PC-Host	932	931
Japan	PC-Host	932	939
Japan	PC-Host	932	290
Japan	PC-Host	932	1027
Japan	PC-Host	932	5026
Japan	PC-Host	932	5035
Taiwan	PC-Host	950	937
Taiwan	Host-PC	937	950
Korea	PC-Host	949	933
Korea	Host-PC	933	949
China	PC-Host	936	935
China	Host-PC	935	936

TrnsDT Resource Files

The TrnsDt API uses a series of resource files that contain the necessary translation tables. These files are supplied on the CD-ROM for SNA Server 3.0 and later.

File name	Description
TRNSDT.DLL	Core global resource used by all TrnsDT conversions
TRNSDTJ.DLL	Core Japanese resource
TRNSDTS.DLL	Core Simplified Chinese (PRC) resource
TRNSDTK.DLL	Core Korean resource
TRNSDTT.DLL	Core Traditional Chinese (Taiwanese) resource
SNADBC.TBL	Japanese double-byte translation tables
SNADBCS.TBL	Simplified Chinese (PRC) double-byte translation tables
SNADBCT.TBL	Simplified Chinese (Taiwanese) double-byte translation tables
SNADBCK.TBL	Korean double-byte translation tables
SNASBC.TBL	Japanese single-byte translation tables
SNASBCS.TBL	Simplified Chinese (PRC) single-byte translation tables
SNASBCK.TBL	Korean single-byte translation tables
SNASBCT.TBL	Traditional Chinese (Taiwanese) single-byte translation tables

TrnsDT API Functions

The TrnsDT API consists of a single function described in this section.

This section contains:

- TrnsDT

TrnsDT

The **TrnsDT** function is called to translate a string from one code page to another.

```
WORD WINAPI TrnsDt(
    PASSSTRUCT far* PassParm
);
```

Parameters

PassParm
Supplied parameter. A pointer to a **PASSSTRUCT** structure containing members that must be supplied as well as members that are returned by the function.

The PASSSTRUCT structure

The PASSSTRUCT structure is defined as follows:

```
typedef struct tagPassParm {
    WORD    parm_length;
    WORD    exit_code;
    WORD    in_length;
    LPBYTE  in_addr;
    WORD    out_length;
    LPBYTE  out_addr;
    WORD    trns_id;
    WORD    in_page;
    WORD    out_page;
    WORD    option;
    WORD    type;
} PASSSTRUCT;
```

Members

parm_length
Supplied parameter. The length of the structure passed, normally set to 26. If the **type** member is not needed, **parm_length** can be set to 24. If the **option** member and the **type** member are not needed, then **parm_length** can be set to 22.

exit_code
Supplied and returned parameter. On entry this member must be set to zero. On return, this member indicates the exit status. Legal values for returned **exit_code** values are as follows:

0

Normal exit code indicating function completed successfully.

1

The requested conversion is not supported.

12

The **exit_code** field was not properly initialized to zero.

128

The last character in the source input string was a DBCS lead byte.

256

The conversion could not be successfully completed since the length of the resulting converted destination string exceeds 65535 bytes.

257

An error occurred when trying to load one and initialize one of the TrnsDTx.DLL files.

in_length

Supplied parameter. Specifies the length of the input source string in bytes.

in_addr

Supplied parameter. A pointer to the buffer containing the source string to be converted.

out_length

Supplied and returned parameter. Specifies the maximum length available for the output translated string in bytes. On return, this member is set to the length of the converted output string on success or the output buffer length needed if the buffer was too small.

out_addr

Supplied parameter. A pointer to the buffer that will contain the output destination string after conversion.

trns_id

Supplied parameter. The conversion identifier, which is always zero.

in_page

Supplied parameter. Specifies the code page of the incoming source string.

out_page

Supplied parameter. Specifies the code page of the output translated string.

option

Supplied and returned parameter if **parm_length** was set to 24 or higher. As a supplied parameter, this specifies a set of options that may be applied to the translation process. Possible values for these options are as follows:

Bits 15-9

Reserved.

Bit 8

Add shift out (SO)/shift in (SI) bytes to the converted output strings.

Bits 3-7

Reserved.

Bit 2

If this bit is set, then convert the input string using the IBM-specified one-byte code table. This option is only valid when converting from code page 932 to one of the following code pages: 037, 290, 930, or 931.

If this bit is zero, then convert the input source string using the conversion table that is created using the SYSCTBL utility.

In case of double-byte characters, always use the conversion table created by the SYSCTBL utility.

The SYSCTBL.EXE file is a utility program included with Host Integration Server 2004 that provides a tool that can be used to create custom conversion tables for use with the **TrnsDT** function.

Bit 1

If this bit is set, then it indicates that the input source string starts with a 2-byte character. Generally, the host data always includes SO/SI control characters in pairs. But when converting part of mixed data strings, it is necessary to start the conversion from a double-byte character without the SO control character. In this case, the data itself does not have adequate information to determine if it is double-byte or not, so bit 1 must be set.

Bit 0

If this bit is set, then it indicates that the input source string contains SO/SI control characters. Bit 8 and bit 0 should be set as follows:

Conversion from PC to host Bit 8=1, bit 0 =0
Conversion from host to PC Bit 8=0, bit 0=1

OP_BLANK_PAD (**0x0008**)

This option forces the output buffer to be filled with blank characters. The character used depends on the parameter type:

SNA_DBCS: from ANSI to EBCDIC - double byte blanks (x4040) are used to fill the buffer, and SI (x0f) is set at the end of the buffer.

SNA_EITHER or SNA_BOTH: single byte blank (x40) are used to fill the buffer.

From EBCDIC to ANSI - single byte blank (x20) are used to fill the buffer, regardless of the parameter type.

OP_TRUNCATE **(0x0010)**

This option forces the function to exit without sending an error when the output buffer overflows.

OP_OMITNULL (**0x0020**)

This value is a no-op.

On return, **option** is set to 4 if the last character was a double-byte character.

type

Supplied parameter if **parm_length** was set to 26. Possible values for this option are as follows:

SNA_DBCS (0x0100)

Always has SO/SI.

SNA_EITHER (0x0200)

Only one set of SO/SI can be present. This can be DBCS or SBCS, but not both.

SNA_BOTH (0x0400)

A mixture of DBCS/SBCS is allowed.

Return Values

The **TrnsDT** function returns zero on success. On failure, possible values returned by this function are as follows:

ERR_FILE_NOT_FOUND

This error is returned if the TrnsDT table files (*.TBL) could not be found. Normally **TrnsDT** uses the conversion tables located in the Host Integration Server\System directory on Microsoft® Windows 2000. If **TrnsDT** cannot find these tables, it searches for them in the current directory.

ERR_INVALID_PARAMETER

This error is returned if a bad value was passed for one or more of the members of the *PassParm* structure. Invalid parameters can include not zeroing the **exit_code** member, passing an **in_length** for the input source string of zero or less or greater than 65535 bytes, passing an **out_length** for the output string buffer of zero or less, passing **in_page** or **out_page** members containing undefined codepage values.

ERR_BUFFER_OVERFLOW

This error is returned if the output buffer is too small for the converted output string. In such cases, the **out_length** member returns with the necessary value in bytes for the output buffer. This error is also returned if the length of the output buffer needed to convert the source string would exceed 65535 bytes.

ERR_MEMORY_ALLOCATE

This error is returned if memory could not be allocated for use by the TrnsDT DLL.

Host Integration Server 2000 Components and NLS Support

The following table lists the conversion methods and types used by the various components in Microsoft® Host Integration Server 2000.

Component	Conversion method	Conversion types	Description
3270 Applet	SNANLS & TrnsDT	SBCS & DBCS	
5250 Applet	SNANLS & TrnsDT	SBCS & DBCS	
Security Integration Service	SNANLS & TrnsDT	SBCS & DBCS	
Host Security		SBCS	User identifiers and passwords are converted from PC to host. Only Latin I code pages supported.
Shared Folders Service	SNANLS	SBCS	Folder names. Files are not converted.
AFTP Service	Host computer		
NetView Alert Service	Proprietary	DBCS	
NetView RunCmd Service	Proprietary	DBCS	
CSV Convert Verb	Proprietary		
ODBC Driver for DB2	SNANLS & TrnsDT	SBCS & DBCS	
OLE DB Provider for DB2	SNANLS & TrnsDT	SBCS & DBCS	
OLE DB Provider for AS/400 and VSAM	SNANLS & TrnsDT	SBCS & DBCS	
VSAM File Transfer	SNANLS & TrnsDT	SBCS & DBCS	
Data Queues ActiveX Control	SNANLS & TrnsDT	SBCS & DBCS	
Host File Transfer ActiveX Control	SNANLS & TrnsDT	SBCS & DBCS	

SNA Server Components and NLS Support

The following table lists the conversion methods and types used by the various components in Microsoft® SNA Server 4.0.

Component	Conversion method	Conversion types	Description
3270 Applet	Proprietary & TrnsDT	SBCS & DBCS	
5250 Applet	Proprietary & TrnsDT	SBCS & DBCS	
Security Integration Service	SNANLS & TrnsDT	SBCS & DBCS	
Host Security		SBCS	User identifiers and passwords are converted from PC to host. Only Latin I code pages supported.
Shared Folders Service	SNANLS	SBCS	Folder names. Files are not converted.
AFTP Service	Host computer		
NetView Alert Service	Proprietary	DBCS	
NetView RunCmd Service	Proprietary	DBCS	
CSV Convert Verb	Proprietary		
ODBC Driver for DB2	SNANLS and TransDT	SBCS & DBCS	
OLE DB Provider for DB2	SNANLS & TrnsDT	SBCS & DBCS	
OLE DB Provider for AS/400 and VSAM	SNANLS & TrnsDT	SBCS & DBCS	
VSAM File Transfer	SNANLS & TrnsDT	SBCS & DBCS	

SNA Print Server Data Filters

The Host Print Service feature of Microsoft® Host Integration Server 2000 and SNA Server provides server-based 3270 and 5250 printer emulation, allowing host applications to print to a LAN printer supported by Microsoft Windows® 2000 Server, Windows NT® Server, and Novell NetWare. This section introduces the SNA Print Server Data Filter API (sometimes referred to as the Print Exit API) that can be used to extend the capabilities of the Host Print Service in Host Integration Server 2000 and SNA Server. The user can provide a print data filter DLL that will be called by Host Print Service when a print job is initiated, when data is sent to the printer, and when the print job is completed. This print data filter DLL can:

- Send data to the printer when a job starts (print a banner page, for example).
- Perform special processing on the data to be printed.
- Send data to the printer upon print job completion (print a trailer page, for example)

This section contains:

SNA Print Server Data Filter

The Host Print Service feature of Host Integration Server 2000 and SNA Server provides server-based 3270 and 5250 printer emulation, allowing host applications to print to a LAN printer supported by Windows 2000 Server, Windows NT Server, and Novell NetWare. This section introduces the SNA Print Server Data Filter API (sometimes referred to as the Print Exit API) that can be used to extend the capabilities of the Host Print Service in Host Integration Server 2000 and SNA Server. The user can provide a print data filter DLL that will be called by Host Print Service when a print job is initiated, when data is sent to the printer, and when the print job is completed. This print data filter DLL can:

- Send data to the printer when a job starts (print a banner page, for example).
- Perform special processing on the data to be printed.
- Send data to the printer upon print job completion (print a trailer page, for example).

SNA Print Server Data Filter API

The user configures the path to the print data filter DLL. This DLL is used by all sessions actively using the Host Print Service. However, the print data filter DLL can specify whether or not it wants a given session's print data passed to it.

The entry points to this DLL are:

PrtFilterAlloc
: Obtains a data buffer in which to pass print data.

PrtFilterFree
: Indicates that a data buffer obtained previously from the DLL is no longer needed and the DLL can free the memory allocated for this resource.

PrtFilterJobData
: Allows the DLL to manipulate print data.

PrtFilterJobEnd
: Informs the DLL that a print job has ended.

PrtFilterJobStart
: Informs the DLL that a new print job has started and enables the DLL to send special data to the Print Server at the start of a job.

A description of the example sequence of calls during an ordinary print job is listed below to illustrate how these functions are normally used.

- **PrtFilterStartJob** is called when a new print job is started. The DLL can return a data buffer with special data that will be sent to the printer (a special banner page or special printer initialization strings, for example) before printing data.
- **PrtFilterFree** is called if special data was sent in the **PrtFilterStartJob** function and indicates that the data buffer used to pass special data can be freed.

The next sequence of function calls is repeated until all of the print data has been sent.

- **PrtFilterAlloc** is called to allocate a data buffer used to pass print data in the subsequent call to **PrtFilterJobData**.
- **PrtFilterJobData** is called to pass print data to the DLL for possible modification. This allows the user DLL the opportunity to manipulate the printer data before it is sent to the printer. If the modified print data to be returned requires a larger data buffer or the DLL needs to use a different data buffer for returning data, the DLL may need to allocate a new data buffer to return this data. The DLL may also choose to free the data buffer used to pass the incoming print data if a different data buffer is used to return modified print data. The **PrtFilterFree** function will not be called with the pointer to the original data buffer if a different data buffer is returned by **PrtFilterJobData**.
- **PrtFilterFree** is called to indicate that the data buffer allocated by **PrtFilterAlloc** for passing incoming data to the **PrtFilterJobData** function can be freed. If a different data buffer was returned by **PrtFilterJobData**, then **PrtFilterFree** would be called to indicate that a data buffer allocated by the DLL used to return modified print data in the **PrtFilterJobData** function can be freed.

The final sequence occurs when all of the print data has been processed.

- **PrtFilterEndJob** is called to indicate the end of the print job and allows the DLL the option to return special data (a trailer page, for example) that should be sent to the printer.
- **PrtFilterFree** is called if special data was sent in the **PrtFilterEndJob** function and indicates that the data buffer used to pass special data can be freed.

PrtFilterAlloc

The **PrtFilterAlloc** function is called to obtain a data buffer from the user filter DLL in which to pass it the print data.

```
void * WINAPI PrtFilterAlloc(
  DWORD BufLen
);
```

Parameters

BufLen
Supplied parameter. Indicates the length of buffer required.

Return Values

The **PrtFilterAlloc** function allocates a memory block of *BufLen* size and returns a pointer to the buffer. This function should return a NULL pointer on failure.

See Also

PrtFilterFree

PrtFilterFree

The **PrtFilterFree** function is called to indicate that a data buffer obtained previously from the DLL is no longer needed and the DLL can free the memory allocated for this resource. This function is called for data buffers returned from calls to **PrtFilterAlloc** as well as buffers that were allocated by the DLL to pass data in the **PrtFilterStartJob**, **PrtFilterJobData**, and **PrtFilterEndJob** functions.

```
void WINAPI PrtFilterFree(
  void *pBuf
);
```

Parameters

pBuf
Supplied parameter. Points to the data buffer that can be be freed.

See Also

PrtFilterAlloc

PrtFilterJobData

The **PrtFilterJobData** function is called to give the user DLL the opportunity to manipulate the printer data before it is printed. This allows the DLL to provide custom processing for print data sent to the print server.

```
void WINAPI PrtFilterJobData(
  void *UniqueID,
  char **pBufPtr,
  DWORD *pBufLen
);
```

Parameters

UniqueID
Supplied parameter. The *UniqueID* value returned by the **PrtFilterJobStart** function to identify a print job.

pBufPtr
The print server passes the print data received from the host to the user DLL for processing in this incoming buffer. The user DLL returns to the print server a pointer to an outgoing buffer of data to be printed. This outgoing buffer pointer can be different from the received buffer pointer because the print data filter DLL can modify the data. Note that in this case **PrtFilterFree** will only be called by the Host Print Service for the outgoing buffer pointer. If necessary, the print data filter DLL must call its own free function on the incoming buffer pointer that was supplied to the **PrtFilterJobData** function. This incoming buffer was allocated by a Host Print Service by a previous call to **PrtFilterAlloc.**

pBufLen
Indicates the length of the data passed in the buffer to the print server and the length of the buffer returned to the print server by the user-provided DLL.

Remarks

The data in the buffer is printable ASCII and/or printer control sequences if these are being sent in the print jobs. The buffer returned by the user DLL does not have to be the same as the buffer passed in. The returned buffer will always be freed by calling **PrtFilterFree** after the data has been spooled. The unique identifier parameter *UniqueID* is the identifier returned from a previous call to the **PrtFilterJobStart** function.

See Also

PrtFilterFree, PrtFilterJobStart

PrtFilterJobEnd

The **PrtFilterJobEnd** function is called to inform the print data filter DLL that a print job is about to end. This allows the DLL to provide custom processing and send special data to the print server at the end of a print job.

```
void * WINAPI PrtFilterJobEnd(
  void *UniqueID,
  char **pBufPtr,
  DWORD *pBufLen
);
```

Parameters

UniqueID
Supplied parameter. The *UniqueID* value returned by the **PrtFilterJobStart** function to identify a print job.
pBufPtr
Returned parameter. Specifies a pointer to a buffer pointer holding additional data to be printed by the print server.
pBufLen
Returned parameter. Pointer to the length of the data provided by the print data filter DLL in the buffer.

Remarks

No data is passed in the buffer, but the user DLL can return print data which will be sent to the printer before the print job is ended.

See Also

PrtFilterJobStart

PrtFilterJobStart

The **PrtFilterJobStart** function is called to inform the print data filter DLL that a new job has just been started. This allows the DLL to provide custom processing and send special data to the print server at the beginning of a job.

```
void * WINAPI PrtFilterJobStart(
  char *SessionName,
  DWORD LUType,
  char **pBufPtr,
  DWORD *pBufLen
);
```

Parameters

SessionName
Supplied parameter. The name of the print session which has just started a print job. The *SessionName* is the same as that configured in using the SNA Print Server Admin tool.

LUType
Supplied parameter. Specifies the printer type. Valid values are LU 1, LU 3, or LU 6.2 printers, represented by an *LUType* value of 1, 3, or 6.

pBufPtr
Returned parameter. Specifies a pointer to a buffer pointer holding additional data to be printed by the print server.

pBufLen
Returned parameter. Pointer to the length of the data provided by the print data filter DLL in the buffer.

Return Values

The **PrtFilterJobStart** function returns a unique identifier (cast to a pointer to a void) if it wants the opportunity to filter the data for this print job.

If the user DLL returns a NULL pointer, it is indicating that it is not interested in filtering this job. No further calls to the user DLL will be made for this print job.

Remarks

No data is passed in the data buffer to the print data filter DLL in this call, but the DLL can return data in *pBufPtr* (for example, a banner page). The data returned from this call should be printable ASCII and/or printer control sequences.

Sample Programs for SNA Print Server Data Filter

The source code for a sample program that illustrates using the SNA Print Server Data Filter API is included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. The sample program illustrates features of the SNA Print Server Data Filter API that can be used to extend the capabilities of the Host Print Service in Host Integration Server 2000 and SNA Server. The sample code illustrates how to write a print data filter DLL that will be called by Host Print Service when a print job is initiated, when data is sent to the printer, and when the print job is completed.

The SNA Print Server Data Filter sample program is located in the \SDK\Samples\SNA\PrnFltr subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\SNA\PrnFltr subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\SNA\PrnFltr subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

This sample program include the following files:

Subdirectory	Description
NTFilter.c	The sample program written in Visual C that illustrates use of the SNA Print Server Data Filter API.
NTFilter.h	An include file with function defines.
NTFilter.def	A DEF file for use by the compiler and linker. Note that exported functions from the SNA Print Server Data Filter DLL must not be decorated.
NTFIlter.rc	A resource file.
Makefile	A command-line Makefile that can be used with nmake. This Makefile can be used with Microsoft® Visual Studio .NET or Microsoft® Visual Studio 6.0.
prnfltr.vcproj	A project file for use with Visual C++ 7.0 and the Visual Studio .NET IDE.

The PrnFltr sample is designed to be built using Microsoft® Visual C/C++ 6.0 or later using the command-line compiler or using the Microsoft® Visual Studio 6.0 or Microsoft® Visual Studio .NET interactive development environment (IDE).

To build the PrnFltr sample using the command-line compiler, set up your build environment as follows:

- Open an MS-DOS Command Prompt window.
- Run VCVARS32.bat (for VS6) or VSVARS32.bat (for VS.NET) from the Visual Studio bin directory(by default, C:\Program Files\Microsoft Visual Studio\VC98\Bin for VS6 or C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools for VS.NET))
- Navigate to SNA\PrnFltr subdirectory, and invoke NMAKE.

To build the PrnFltr sample using the Visual Studio .NET IDE, start Microsoft Visual Studio .NET 7.0 and open the appropriate Visual C++ 7.0 project file (SNA\PrnFltr\ntfilter.vcproj) from the **File** menu. Select a configuration and build the sample from the **Build** menu. Each VC7 project file has two configurations, one for a DEBUG build and one for a RETAIL build.

Device Interface Specification Drivers

This section of the documentation is intended for OEMs and adapter vendors who are developing their own SNALink software to work with Microsoft® Host Integration Server 2000 and Microsoft SNA Server 4.0.

This section contains:

About the SNADIS Guide

This guide is intended for OEMs and adapter vendors who are developing their own SNALink software to work with Microsoft® Host Integration Server 2000 and Microsoft SNA Server 4.0.

SNALink software for use on Microsoft Windows® 2000 must be written as a Windows 2000 device driver. See the Windows 2000 Device Driver Kit (DDK) for information on writing Windows 2000 device drivers.

Before using this section, you should be familiar with SNA Server concepts. This section provides the following information:

- Internal concepts of Host Integration Server 2000 and SNA Server 4.0 that are required to integrate new communications adapters into the server environment.
- Definitions of the interfaces used by Host Integration Server 2000 and SNA Server 4.0 to communicate with SNALinks.
- Information on using the configuration and diagnostics features included in Host Integration Server 2000 and SNA Server 4.0.
- Instructions for compiling and linking the SNALink support software.

The network operating systems currently supported by Host Integration Server 2000 and SNA Server 4.0 include Microsoft LAN Manager (as implemented in Microsoft Windows 2000, Windows NT®, Windows 98, and Windows 95), Novell NetWare, Banyan VINES on Windows NT 4.0, and TCP/IP. Future versions of Host Integration Server may support other network operating systems. You are advised to develop link support that is independent of the network operating system in order to take advantage of this support in future versions.

SNADIS Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide provides the programmatic techniques and procedures for creating SNALink applications.

This section contains:

SNALink Concepts in Host Integration Server and SNA Server

This section describes some key concepts used in Microsoft® Host Integration Server 2000 and SNA Server. Since the purpose of this document is to enable original equipment manufacturers (OEMs) and adapter vendors to develop link support software (an SNALink*)* to integrate their hardware adapters into a Host Integration Server 2000 or SNA Server system, only the relevant parts of the Host Integration Server 2000 and SNA Server architecture are described.

This section contains:

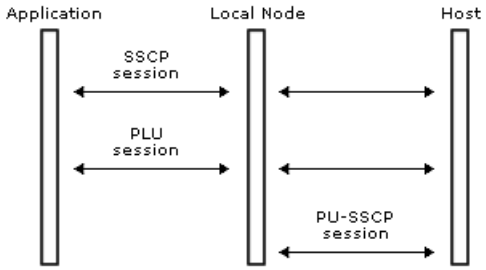
Overview of SNALink

A Microsoft Host Integration Server 2000 or SNA Server SNALink must implement an SNA-compatible data transport mechanism capable of connecting the local type 2.1 node to remote host (PU4/5) and/or peer (PU2.1) systems.

The local node provides the SNA layers of path control, transmission control, data flow control, and logical unit (LU) services. The following figure shows an example of a Host Integration Server 2000 system.

The local node uses the data link control interface (DLC) to communicate with an SNALink. This interface is defined in The Data Link Control Interface. The SNALink and the DLC driver are responsible for transferring data between the path control layer of the node and the DLC adapter.

The routing of messages that flow between Host Integration Server 2000 or SNA Server components is handled by the SnaBase and DMOD (dynamic access module) components. Refer to The SNALink Interface for details of how to send and receive messages.



SNALink Configuration and Management

The configuration information for a Microsoft Host Integration Server 2000 or SNA Server system is stored in two forms:

- A centralized configuration file containing details of logical units (LUs), physical units (PUs), and connections.
- Entries in the Microsoft Windows® 2000, Windows NT®, or Windows 95/98 registry containing configuration information for the SNALinks supported on that machine. This information contains a few parameters required by Host Integration Server 2000 or SNA Server and any other parameters that independent hardware vendor (IHV) code may require.

A Host Integration Server or SNA Server SNALink is defined when a Host Integration Server or SNA Server system is installed. A SNALink can support only one physical connection from the server. If a single adapter is capable of supporting multiple physical connections, Host Integration Server or SNA Server requires multiple SNALinks to be configured.

To reconfigure a server's SNALink support (for example, after installing a new adapter), the administrator uses either the Windows Network Control Panel applet or the Host Integration Server or SNA Server setup program. For further details of how this operates, refer to Setup Information.

All other configuration of a Host Integration Server or SNA Server system is performed using SNA Manager. Refer to the *Administrator's Reference* for further details. As part of the configuration process, logical connections to remote PUs are associated with one or more SNALinks.

All configured SNALinks are automatically started when the Host Integration Server or SNA Server system is started. At this stage, the SNALink performs any initialization required and then waits for instructions from local nodes.

When a connection is activated, either from SNA Manager or automatically (for example, in response to a 3270 user's request for a session with a remote host), the SNALink receives an Open(LINK) message from the local node. The SNALink should then perform whatever action is required to initiate that connection. This can involve dialing a telephone number for a switched Synchronous Data Link Control (SDLC) connection or bringing up level 2 on an X.25 link and sending a CALL packet.

If the IHV wants the same physical adapter to be available for use by multiple SNALinks (for example, a dumb SDLC card can be used to communicate using SDLC or X.25 protocols), the SNALink should not attempt to access the hardware until it has received an Open(LINK) message from the local node.

Structure of Host Integration Server and SNA Server Components

The components of Microsoft Host Integration Server 2000 or SNA Server are local nodes, SNALinks, 3270 emulators, and so on. This section introduces the structure of these components and explains terms used to refer to the structure.

This section contains:

The Role of the Base

The Base is a part of each Host Integration Server 2000 or SNA Server component, such as a local 2.1 node or an SNALink, that provides the operating environment for that component. It passes messages between components and provides functions common to all components, such as diagnostic tracing.

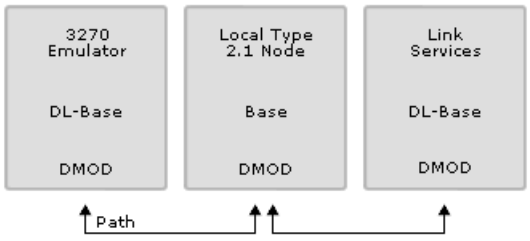
This document is concerned with the link Base, which is the type of Base used by Host Integration Server or SNA Server SNALinks. The Base has entry points for initialization, sending messages, receiving messages, and termination.

Localities and DMODs

A Base and its components (that is, a Host Integration Server 2000 or SNA Server executable program) is called a locality. The Host Integration Server or SNA Server system therefore consists of one or more communicating localities (all the running Host Integration Server 2000 or SNA Server executable programs within the LAN Manager domain). For each Host Integration Server or SNA Server system, there is a central configuration file. In addition, each Host Integration Server or SNA Server machine maintains configuration information about the SNALinks it supports (see Diagnostics).

In a system such as Host Integration Server or SNA Server, where the number of localities and their types are not configured in advance, the relationships between the localities are set up dynamically as individual localities come and go. Localities that can enter and leave a system in this way are called dynamic localities.

Dynamic localities communicate using the DMOD (dynamic access module) component, which provides the communications facilities needed to pass messages between the Bases. This is illustrated in the following figure.



This diagram shows a system consisting of three dynamic localities. Dynamic localities can enter or leave this system at any time.

Component Localities

SNALinks can enter dynamically into a Host Integration Server 2000 or SNA Server system. The SNALink, in conjunction with the Base, acts as a whole locality and communicates with the other localities in the system using a DMOD.

The SNALink Interface describes the interface to the Base and the DMOD that allows an SNALink (or any other CS component) to participate in a Host Integration Server 2000 system.

Partners

For Host Integration Server 2000 or SNA Server components and applications to communicate with each other, it must be possible to identify a partner within a locality. A partner is an addressable component of a locality; that is, code to which messages can be sent. In a Host Integration Server or SNA Server system, there is generally only one partner within a locality (such as an SNALink or the 3270 emulation program); however, separate functions within the local 2.1 node (such as the 3270 and APPC functions) can be considered to be separate partners.

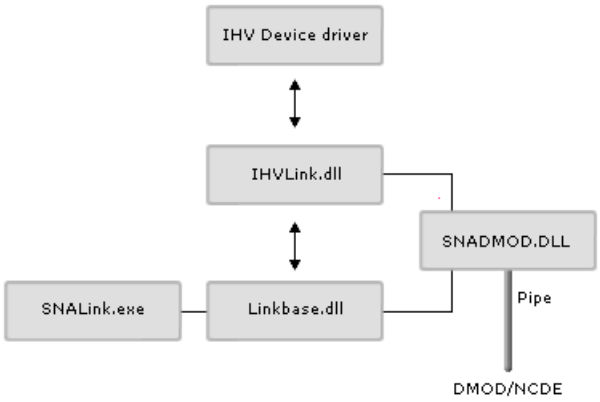
SNALink Structure

A Host Integration Server 2000 or SNA Server SNALink consists of:

- The link-specific protocol code provided by the IHV
- A Base
- A DMOD

The DMOD, Base, and the IHV link-specific component of a Host Integration Server or SNA Server SNALink are implemented as dynamic-link libraries (DLLs). The executable component SNALINK.EXE is used to start an SNALink. This component determines from the Host Integration Server or SNA Server configuration information which link support DLL (for example, IHVLINK.DLL) is required for the SNALink and dynamically loads it before entering the Base scheduler.

The executable structure of an SNALink is shown in the following figure.



Messages

Messages are used to pass data between partners in the Microsoft Host Integration Server 2000 or SNA Server system. This section provides information about message structure and formats.

This section contains:

Overview of Message Formats

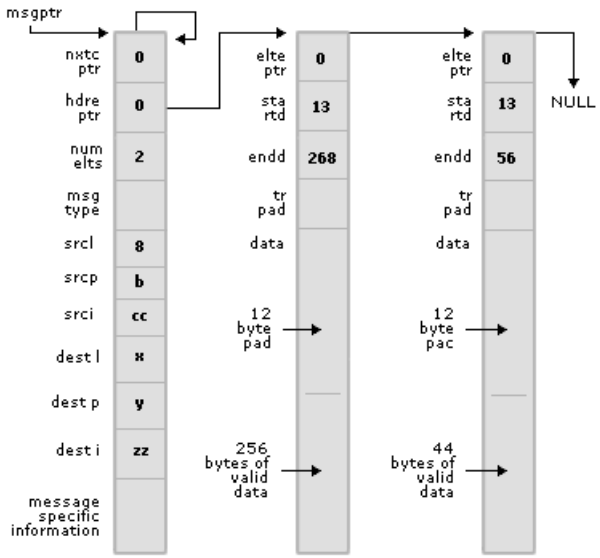
A message always contains fixed-format header information such as a message type and addressing information. It can also contain other header information specific to a particular message type (such as the message subtype) and an indefinite amount of extra data.

Messages are saved in buffers that consist of one header and zero or more elements:

- The header contains the fixed-format information and a pointer to an element. (This pointer will be NULL if there are no elements associated with the message.)
- An element contains any extra data for a message and a pointer to another element if the data continues into another element.

Buffer headers and elements are regarded as contiguous (8-bit) byte sequences. Messages of any length can be built up by chaining sufficient elements to a header.

The following figure illustrates a typical message with two elements. The individual fields in the header and elements are explained in the following topics.



Buffer Header Format

This topic lists the common fields that always occur at the start of a buffer header. These are followed by further fields specific to the particular message; see Message Formats for details of individual message formats.

Field	Type	Description
PTRBFHDR	nxtqptr	When the buffer is in a queue, this field points to the header of the next buffer in the queue (NULL if it is the last buffer in the queue). When the buffer is not in a queue, this field points to itself; the Host Integration Server 2000 buffer management routines use this to check for buffer corruption.
PTRBFELT	hdreptr	Pointer to the first buffer element in the associated chain of buffer elements; NULL if the message consists only of a buffer header.
CHAR	numelts	Number of buffer elements chained from the header; zero if the message consists only of a buffer header.
CHAR	msgtype	Message type. See individual message descriptions in Message Formats.
CHAR	srcl	Source locality. See LPI Addresses.
CHAR	srcp	Source partner. See LPI Addresses.
INTEGER	srci	Source index. See LPI Addresses.
CHAR	destl	Destination locality. See LPI Addresses.
CHAR	destp	Destination partner. See LPI Addresses.
INTEGER	desti	Destination index. See LPI Addresses.

```
PTRBFHDR  nxtqptr;
PTRBFELT  hdreptr;
CHAR      numelts;
CHAR      msgtype;
CHAR      srcl;
CHAR      srcp;
INTEGER   srci;
CHAR      destl;
CHAR      destp;
INTEGER   desti;
};
```

Members

nxtqptr
When the buffer is in a queue, this field points to the header of the next buffer in the queue (NULL if it is the last buffer in the queue). When the buffer is not in a queue, this field points to itself; the Host Integration Server 2000 buffer management routines use this to check for buffer corruption.

hdreptr
Pointer to the first buffer element in the associated chain of buffer elements; NULL if the message consists only of a buffer header.

numelts
Number of buffer elements chained from the header; zero if the message consists only of a buffer header.

msgtype
Message type. See individual message descriptions in Message Formats.

srcl
Source locality. See LPI Addresses.

srcp
Source partner. See LPI Addresses.
srci
Source index. See LPI Addresses.
destl
Destination locality. See LPI Addresses.
destp
Destination partner. See LPI Addresses.
desti
Destination index. See LPI Addresses.

> **Note** Fields that occupy two bytes, such as **opresid** in the Open(LINK) request, are normally represented with the arithmetically most significant byte in the lowest byte address, irrespective of the normal orientation used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. However, the following fields are exceptions:

- The srci and desti fields in buffer headers are stored in the local format of the application that assigns them (only the assigning application needs to interpret these values).
- The startd and endd fields in elements are always stored in low-byte, high-byte orientation (the normal orientation of an Intel processor).

Buffer Element Format

This topic lists the common fields that always occur at the start of a buffer element. The **dataru** field contains information specific to the particular message; see Message Formats for details of individual message formats.

Field	Type	Description
PTRBFELT	hdreptr–>elteptr	Pointer to next buffer element in the chain; NULL if this element is the last or only element in the chain.
INTEGER	hdreptr–>startd	Start of valid data in this element. The index into *dataru* of the first byte of valid data.
INTEGER	hdreptr–>endd	End of valid data in this element. The index into *dataru* of the last byte of valid data.
CHAR	hdreptr–>trpad	Pad byte (reserved).
CHAR[SNANBEDA]	hdreptr–>dataru	An array of characters that contains the data for this element. Note that the valid data might not occupy the whole of the element; *startd* and *endd* (see above) give the indexes into this array of the start and end of the valid data. The constant SNANBEDA is defined in SNA_DLC.H as 268.

```
PTRBFELT        hdreptr–>elteptr;
INTEGER         hdreptr–>startd;
INTEGER         hdreptr–>endd;
CHAR            hdreptr–>trpad;
CHAR[SNANBEDA]  hdreptr–>dataru;
};
```

Members

hdreptr–>elteptr
Pointer to next buffer element in the chain; NULL if this element is the last or only element in the chain.

hdreptr–>startd
Start of valid data in this element. The index into **dataru** of the first byte of valid data.

hdreptr–>endd
End of valid data in this element. The index into **dataru** of the last byte of valid data.

hdreptr–>trpad
Pad byte (reserved).

hdreptr–>dataru
An array of characters that contains the data for this element. Note that the valid data might not occupy the whole of the element; **startd** and **endd** give the indexes into this array of the start and end of the valid data. The constant SNANBEDA is defined in SNA_DLC.H as 268.

The following information will help you to interpret the message formats:

- Fields that occupy 2 bytes are represented with the arithmetically most significant byte in the lowest byte address, irrespective of the normal orientation used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. The exceptions to this are the **startd** and **endd** fields in elements, which are always stored in low-byte, high-byte orientation (the normal orientation of an Intel processor).
- The offsets indicated by the startd and endd fields are expressed in terms of the first byte of dataru being offset 1; the first byte of valid data is at dataru(startd–1). For example, if startd is 11 and endd is 18, then dataru begins with 10 bytes that are not valid data, followed by 8 bytes of valid data.

In the example message format illustrated in Overview of Message Formats, each element has a **startd** of 13, indicating 12 bytes of padding before the start of the valid data. This leaves room for 256 bytes of data, and hence the element data (300 bytes long in this example) requires two elements.

LPI Connections

Partners communicate by passing messages to each other. If two partners wish to communicate with each other, an LPI connection is set up between the two partners. Messages then flow between the partners over this connection. The term "LPI connection" is explained in LPI Addresses; note that this is not related to the Microsoft Host Integration Server 2000 concept of a connection between the local node and a remote system.

This section contains:

- Paths and DMODs
- LPI Addresses
- Making Connections

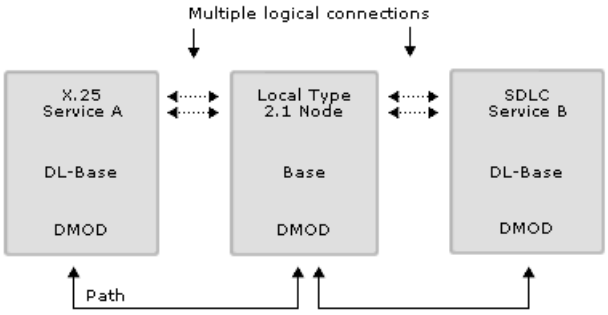
Paths and DMODs

Dynamic access modules (DMODs) are responsible for the communication between localities. When the DMODs in two localities can successfully pass messages between them, a path is said to exist between the two localities. A path must exist between two localities before a connection can exist between partners in those localities.

In Host Integration Server 2000 or SNA Server, a path is implemented using reliable LAN connections (named pipes, SPX, TCP, AppleTalk, VINES IP)—one LAN connection for each path. When the two localities are on the same PC, a local pipe is used; this is implemented using shared buffers to increase performance, but is used by the application in exactly the same way as communication with a remote locality.

The DMOD provides communication between dynamic localities and provides guaranteed in-order delivery of messages flowing over paths between localities. If the DMOD loses its path to another locality, it informs the Base.

The following figure illustrates the paths and connections between a Host Integration Server or SNA Server local node and two SNALinks. X.25 service A has two connections to the local node (one for each of two virtual circuits); SDLC service B has one connection to the local node.



LPI Addresses

An LPI address is used to identify each end of a connection. It has three components: locality (L), product (P), and index (I).

- **Locality** is a 1-byte identifier that uniquely identifies a locality within a system. This locality corresponds to a Host Integration Server 2000 or SNA Server component (local node, SNALink, 3270 emulator, and so on).
- Product is a 1-byte identifier for the type of service. Each type of service has a unique value. A Host Integration Server or SNA Server local type 2.1 node has a defined value of 0x11. A Host Integration Server or SNA Server emulator has a defined product identifier of 0x12. A Host Integration Server or SNA Server link service (X.25. SDLC, Token Ring, Ethernet, Twinax, or Channel, for example) has a defined value of 0x16.
- Index is a 2-byte identifier that uniquely identifies a logical entity within the product. The meaning and use of this field is defined by the communicating services; it is used to distinguish multiple connections between the same services (for example, to identify one of many virtual circuits available from an X.25 SNALink). The value of zero should not be used as an index. Applications must assign unique index values for every active LPI connection within the node.

A message flowing over a connection carries a pair of LPIs, identifying the source and destination of the message. These are the source LPI and destination LPI of the message; together they identify the connection on which the message is flowing.

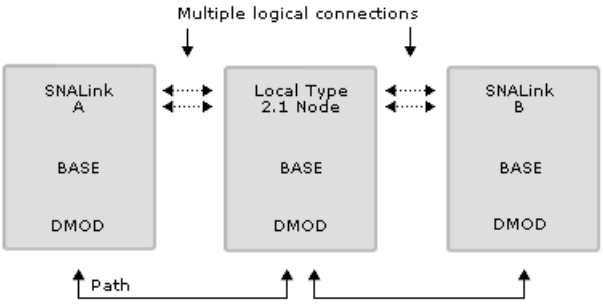
Note that more than one connection can exist between any pair of services. The Index values are then used to distinguish the connections. For example, in communications between the local node and an SNALink, the L and P values identify the message as being DLC data for that local node, and the I value indicates which connection the data is intended for.

The LPIs are assigned by a combination of the products and the DMODs when the connection is opened, as described in Making Connections.

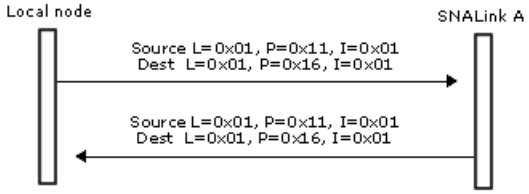
Because they are assigned dynamically for each component, the L values are not the same across a whole system. For example, a local 2.1 node locality could be known as locality 4 to one SNALink locality and locality 6 to a second SNALink locality. However, from the viewpoint of any locality there exists a unique L value for each remote locality within which a path exists; this L value is used as an index into an internal table that identifies the path to that locality.

The following three figures show an example of the L values that could be used between the components shown in Paths and DMODS, and examples of the LPI values that would be used by the local node on messages flowing between the components.

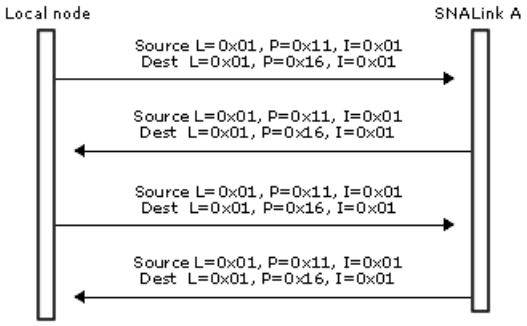
The first illustration shows sample L values.



The following illustration shows L values specified on messages between the local node and SNALink A.



The following illustration shows LPI values specified on messages flowing on two different connections between the local node and SNALink A.



The Base is called by any piece of code that wishes to send a message. It uses the destination L value on the message to determine where to send it. When the message gets to the remote locality, the Base in that locality routes it to the appropriate service if the locality contains more than one service.

Making Connections

Before messages can flow across connections, the connections must be established, or opened. This is necessary because a service does not initially know the LPI address of the service with which it wishes to communicate; indeed, there may not even be a suitable service for it to communicate with.

When a local node wishes to communicate with an SNALink, it attempts to open a connection by sending an Open(LINK) request to the SNALink. This message will have LPI values already set up by the Base, which the SNALink should save for referencing the connection in the future.

The DLC interface does not permit the SNALink to issue an Open request.

The SNALink Interface

The SNALink interface specifies how an IHV link DLL fits into the SNA link service architecture provided by the Base/DMOD interface. This section describes the SNALink interface, the entry points that an IHV link DLL can call, and those functions that a link service must provide to the Base/DMOD interface. These entry points allow messages to be sent to and received from the local 2.1 node.

This section contains:

Process Structure and Scheduling

The primary thread of execution within a Microsoft® Host Integration Server 2000 SNALink is under the complete control of the Base. The Base schedules the SNALink by calling predefined entry points, which the IHV link support code must provide.

The IHV link support code can spawn extra threads of execution. However, the Base is not reentrant. The IHV code must ensure that only a single thread is executing within the Base at any moment in time.

The recommended SNALink structure uses the dispatcher to handle messages received from the local node and the work manager to process data received from the link. These routines and the way in which they are scheduled are described in The Dispatcher and The Work Manager respectively.

SNALink Initialization

When the SNALink is loaded into memory, the Base/DMOD performs all initialization required by the Microsoft® Host Integration Server 2000 system, including announcing availability of the new SNALink to other Host Integration Server 2000 components.

When this has been completed, the Base/DMOD calls the SNALinkInitialize function, which must be provided by the IHV link support code.

SNALinkInitialize is called with a parameter that is a handle to the global Base event. This handle should be saved by the SNALink and used to signal the Base when an event occurs (for example, when data is received from the link).

The **SNALinkInitialize** function should also:

- Read in the Host Integration Server 2000 configuration information for the SNALink (see SNALink Configuration Information for details).
- Set up any required data structures.
- Register with the driver that provides the support for the hardware adapter, initializing this if necessary.

If initialization fails for any reason (for example, if an associated driver is not installed), the function should report the failure to the administrator by calling **SNAReportStatus**.

SNALink Termination

When a critical error occurs, forcing abnormal termination of the SNALink, the IHV code must ensure that all active connections are cleanly terminated, using whatever protocols are appropriate for the link type in use. (For example, an X.25 SNALink would send a CLEAR packet on all active VCs and possibly take down level 2.)

This should be performed using the process detach facility of the **DLLEntryPoint** function.

Sending Messages

The SNALink should build a message in a buffer and then call the Base to send it. The message contains source and destination LPIs, which are set up when the connection is opened; see LPI Connections for more information.

The SNALink can either obtain a new buffer to contain the message to be sent (using SNAGetBuffer) or reuse one in which it has previously received a message. The application is responsible for any buffer that it has obtained or in which it has received a message; it must either use (or reuse) the buffer to send a message or release it (using SNAReleaseBuffer). If a buffer to be reused does not contain the correct number of elements for the message to be sent, the application can obtain additional elements (using SNAGetElement) or release existing ones (using SNAReleaseElement). It is the application's responsibility to maintain the **numelts** field in the message header.

The function used to send a message to the node is SNASendMessage.

The Dispatcher

Whenever a Base event occurs, the Base calls the link support code dispatcher function SNALinkDispatchProc to handle the event. The term Base event in this context means:

- A message arriving from a local 2.1 node.
- A Base timer tick occurring—this relatively slow event happens approximately every five seconds.
- Losing contact with a local 2.1 node (for example, the machine being powered down).

The **SNALinkDispatchProc** function should examine parameters passed to it by the Base to determine why it has been called (see Sample Code for SNALinkDispatchProc) and call an appropriate function to handle the event. When the event has been processed, control returns to the Base scheduler.

Receiving Messages

The Base calls the SNALink dispatcher function SNALinkDispatchProc when a message is available for it.

Note that after the application receives a message it is responsible for the buffer in which the message was received; it must either reuse the buffer to send a message (using SNASendMessage) or release it (using SNAReleaseBuffer). If the buffer to be reused does not contain the correct number of elements for the message to be sent, the application can obtain additional elements (using SNAGetElement) or release existing ones (using SNAReleaseElement).

The Work Manager

When no work is currently outstanding, the Base thread of execution sleeps, waiting for an event or for a maximum period of five seconds. SNALinks should signal the Base when an event occurs (such as data arriving on the link) by setting the Base global event. A handle to this event is passed on the SNALinkInitialize call.

When the Base is rescheduled, it calls the SNALink work manager function SNALinkWorkProc. This function should handle any link events that have occurred.

A common use of this function is in an SNALink where there is a single thread that handles the protocol of the link and also multiple threads suspended on synchronous calls to a driver read function. When data is received from the link, it is placed on an internal queue, and the driver sets the global Base event. This causes the Base to be scheduled, and **SNALinkWorkProc** is called. **SNALinkWorkProc** then dequeues messages and passes them to the Base to be sent to the local node.

Base/DMOD and SNALink Entry Point Summary

The following table shows entry points divided into the categories SNALink, buffer management, and Base/DMOD, and listed in alphabetic order within each category.

SNALink entry points

SNALinkDispatchProc	Dispatcher.
SNALinkInitialize	Initialize SNALink.
SNALinkTerminate	Terminate SNALink.
SNALinkWorkProc	Work manager.

Buffer management entry points

SNAGetBuffer	Get buffer.
SNAGetElement	Get buffer element.
SNAReleaseBuffer	Release buffer.
SNAReleaseElement	Release buffer element.

Base/DMOD entry points

SNAGetLinkName	Get the name of the SNALink.
SNASendAlert	Send a preformatted NMVT alert to NetView.
SNASendMessage	Send a message to the node.

The following functions are defined in SNALink Configuration Information:

SNAGetConfigValue	Get a named item of configuration information.
SNAGetSystemInfo	Get Microsoft® Host Integration Server 2000 system information.

Note Standard calling conventions [WINAPI] are used for all entry points including those provided by the IHV SNALink.

The format of buffer headers and elements is described in Messages; the formats of individual messages contained in buffers are defined in Message Formats.

Sample Code for SNALinkDispatchProc

This section contains outline source code for the link dispatcher function SNALinkDispatchProc.

```
/*********************************************************************/
/* Firstly, include the SNA Server header files                      */
/*********************************************************************/
#include <sna_dlc.h>
#include <sna_cnst.h>
#include <trace.h>

/*********************************************************************/
/* The link dispatcher routine - SNALinkDispatchProc                 */
/*********************************************************************/
VOID SNALinkDispatchProc (msgptr, function, locality)
PTRBFHDR msgptr;
INTEGER  function;
INTEGER  locality;
{
  INTEGER     discard_buff;
  COM_ENTRY("Ldisp");
  if (msgptr != NULL)
  {
    TRACE4()"received message from local node"));
    discard_buff = FALSE;
    switch (msgptr->msgtype)
    {
      case OPENMSG:
             /* process the OPEN message */
             break;
      case CLOSEMSG:
             /* process the CLOSE message */
             break;
      case DLCDATA:
             /* Data to be sent on link */
             break;
      case DLCSTAT:
             /* Switch on the sub-type of the message */
             switch (msgptr->dshdr.dstype)
             {
               case STRESRCE :
                      /* call flow control processor */
                      break;
               case DLCSDXID:
                      /* call XID processor */
                      break;
               default:
                      discard_buff = TRUE;
                      break;
             }
             break;
      default:
             discard_buff = TRUE;
             break;
    }
    if (discard_buff)
    {
      /* message has not been processed by us, so simply discard */
      SNAReleaseBuffer(msgptr);
      msgptr = NULL;
    }
  }
  else if (function == SBLOST)
  {
    /* Lost contact with local node 'locality'  */
    /* Terminate all connections on this node (matching destl-value) */
```

```
    }
    else if (function == SBTICK)
    {
      /* 5 second timer tick */
    }
    COM_EXIT;
  }
```

SNALink Configuration Information

The configuration information for all SNALinks on a machine is stored hierarchically, referenced by the SNALink name, as shown in the figure below.

The entry for each SNALink must include certain fields that are required by the Microsoft® Host Integration Server 2000 system. These are:

Required Fie lds	Description
TYPE	The type of the SNALink. Acceptable values for TYPE are: SDLC, X25, TOKENRING, DFT, TCPIP, FRAMERELAY, CHAN NEL, ISDN, ETHERNET.
LINKMODULE	The name of the IHV DLL that provides the protocol code.

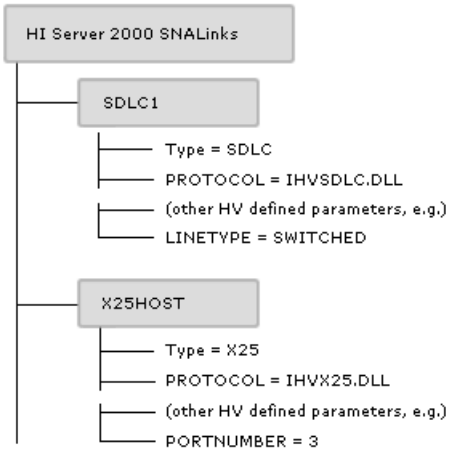
The remainder of the configuration information consists of entries of the form PARAMETER = VALUE. Parameters can be set to either an integer or a string.

Examples of possible parameters that may be required by an SNALink are:

- PortNumber = 3
- LineType = SWITCHED
- L3PacketSize = 128
- T1Timeout = 30

Note that to support more than one port on a multiport adapter, you must define multiple SNALinks. It is not possible to configure a single SNALink to support more than one physical link.

The following illustration shows a sample configuration for a machine with two SNALinks—SDLC1 and X25HOST.



The configuration information is accessed using Host Integration Server 2000-provided API calls.

The IHV Setup utility must write the configuration information for each SNALink supported. Refer to Setup Information for details of how this should be performed.

This section contains:

- Accessing Configuration Information

Accessing Configuration Information

The SNALink uses the following calls to obtain its configuration information:

SNAGetConfigValue	Returns the value of a named configuration parameter.
SNAGetSystemInfo	Returns general information on the version of Microsoft SNA Server currently running, such as the release level, and the network operating system.

If the return code from **SNAGetConfigValue** indicates that the specified configuration parameter is not available, or if the information returned is invalid, it is the SNALink's responsibility to decide what action to take. If appropriate, an error message could be logged; see Diagnostics for more information on error logging.

It is strongly recommended that the SNALink read all required configuration parameters at initialization time (when SNALinkInitialize is called by the Base). This will safeguard against the configuration information changing while the link service is running.

> **Note** Standard calling conventions [WINAPI] are used for all entry points including those provided by the IHV SNALink.

The Data Link Control Interface

The data link control interface (DLC) defines the interface between the local 2.1 node and an SNALink. The DLC interface is defined in terms of the messages that are sent across the interface. Note that this is logically distinct from the definition of the Base/DMOD interface, which defines the API used to send messages between two components in Microsoft® Host Integration Server 2000 (for example, between the local node and an SNALink).

DLC messages are exchanged between the local node and an SNALink across LPI connections (see Structure of Host Integration Server and SNA Server Components).

The local node uses the DLC interface to:

- Activate DLC connections.
- Exchange format 0 or format 3 XIDs for station activation.
- Exchange DLC information frames.
- Handle DLC error notification.

This section contains:

Supported Configurations

The 2.1 node supports the full range of station roles:

- Primary DLC stations
- Secondary DLC stations
- DLC station role negotiation

For SDLC (synchronous data link control) connections, the node allows:

- Leased lines configured as:
    - Secondary point-to-point.
    - Primary point-to-point or multipoint.
    - Negotiable point-to-point.
- Switched lines (point-to-point only) with:
    - Remote PU identification through XID exchange.
    - Auto dial (with suitable hardware support).
    - Incoming call support.

For X.25 and 802.2, the node also supports:

- Multiple connections over one physical link.
- Incoming calls with validation of caller's address.

In addition, for X.25, the node supports permanent and switched virtual circuits (PVCs and SVCs).

Opening a Connection

The 2.1 node is capable of supporting multiple connections through one or more SNALinks. For each connection, the node opens two LPI connections to the SNALink:

- LINK LPI connection to handle activation and deactivation of the connection.
- STATION LPI connection to transfer data to and from the remote station.

The one exception to this rule is the case of primary multipoint connections where there is a single LINK LPI connection and multiple STATION LPI connections. This special case is described in SDLC Multipoint Connections.

The following messages flow over the DLC interface and are used to activate a connection to a remote station.

Open(LINK) Request	Flows from node to DLC over LINK connection.

- Opens the LINK LPI connection between the node and the SNALink.
- Provides configuration data for the SNALink.
- Provides link connection data such as token-ring address for the remote station.

Open(LINK) Response	Flows from DLC to node over LINK connection.

- Reports whether the SNALink has accepted the **Open(LINK) Request**.
- Returns certain link-specific configuration parameters to the local node.
- Can be an OK Response or an Error Response.

Request-Open-Station	Flows from DLC to node over LINK connection.

- Passes an XID received from the SNALink up to the node.
- Indicates that the SNALink has received a mode setting command, such as SNRM over SDLC, or SABME over 802.2.

Send-XID	Flows from node to DLC over LINK connection.

- Passes an XID from the node to the SNALink to be sent out over the link to the remote station.

Open(STATION) Request	Flows from node to DLC over STATION connection.

- Opens the STATION LPI connection between the node and the SNALink.
- Specifies certain station-specific configuration information.

Open(STATION) OK Response –or– Open(STATION) Error Response	Flows from DLC to node over STATION connection.

- Acknowledges Open(STATION) Request.

Station-Contacted	Flows from DLC to node over STATION connection.

- Informs the local node that the link is now ready for data transfer.

The use of these messages in activating various types of connections is described throughout the rest of this section. The format of the messages is given in Message Formats.

The name of the Request-Open-Station message is historical. In earlier versions of the DLC interface, the higher-level software (such as the local node) always sent an Open(STATION) Request in response to this message—hence the name Request-Open-Station. However, now that multiple XIDs can be exchanged before the link is activated, the Open(STATION) Request is only sent at the end of the XID exchange.

The Request-Open-Station message now has two distinct semantic meanings:

- A Receive-XID
- A Receive-Set-Mode

This section contains:

- Opening the LINK LPI Connection
- Activating a Host Connection
- Activating a Peer Connection
- Opening the STATION LPI Connection
- Node Identification and Signaling Information
- XID Retries
- Multiple Connections

Opening the LINK LPI Connection

The local node attempts to activate a connection:

- During system initialization if the connection is configured as initially active.
- If the system administrator manually activates the connection.
- If a 3270 or LU 6.2 session is requested when there is no active connection to support the LU, and the connection is configured to be activated on demand.

For each connection to be activated, the local node opens a LINK LPI connection by sending an Open(LINK) Request to the SNALink. This message contains configuration data such as:

- SDLC line type: leased, switched.
- Operational role: primary, secondary, or negotiable.
- Time-out values.
- Retry limit values.
- Line speed.
- Half-duplex/full-duplex.
- 802.2 remote service access point (SAP) address.
- X.25 facility data.

For incoming calls, the local node primes the SNALink by opening the LINK LPI connection, but does not perform an activation sequence at this stage (see Incoming Call Support).

The local node also inserts the first XID frame to be used (where applicable) and link connection data to be used on a switched link. The link connection data can be:

- A telephone number for a manual or auto-dial modem (in this case, the SNALink software could dial the required number or send a message to the operator specifying the number to be dialed).
- The media access control (MAC) address of the remote station.
- The X.25 remote data terminal equipment (DTE) address.

Finally, the **Open(LINK)** contains various time-out values that should be used by the SNALink when setting up protocol timers. See Open(LINK) Request and Open(LINK) Response.

The SNALink should return an Open(LINK) OK Response if:

- Its internal control blocks are successfully initialized.
- Its device driver has installed correctly.
- Its link hardware is successfully initialized.

The SNALink should not wait for an end-to-end connection before giving an **Open(LINK) Response**.

If the SNALink has successfully initialized, it should return an Open(LINK) OK Response immediately, supplying the required link-specific configuration information to the node (such as the maximum BTU size it can support). The local node will use this information during XID negotiation with the remote station.

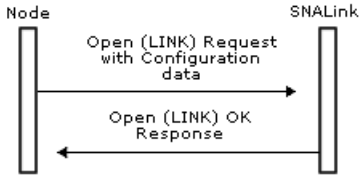
If the SNALink cannot initialize successfully, it responds with an Open(LINK) Error Response containing an error code. The error is logged and the local node notifies the system operator before retrying the link activation.

If an XID is supplied on the Open(LINK) Request, this should be sent when the end-to-end connection is established for a primary or negotiable link. Note that the supplied XID can be a NULL XID, which has a zero length. Hence, it is important that the OPINIXID field is examined rather than checking for a zero XID length. An XID will be supplied for all connections except primary leased connections (which could be multipoint).

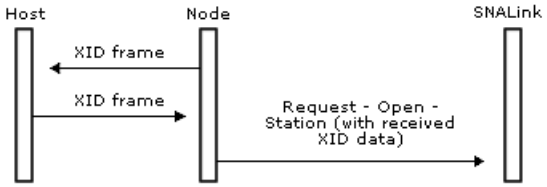
When an SNALink receives an XID frame from the remote station, it is passed to the local node in a Request-Open-Station message on the LINK LPI connection.

If the SNALink fails to receive any frames from the remote station, it generates an Outage message as described in Closing a Connection.

The following figure shows the Open(LINK) Request and Open(LINK) Response, followed by an exchange of XIDs.



End-to-end connection established



Activating a Host Connection

A host connection can be activated over a leased SDLC line, X.25, 802.2, or a switched SDLC line. This section describes the activation procedures for each type of connection.

This section contains:

Leased SDLC Line (No XIDs Exchanged), Channel Adapter

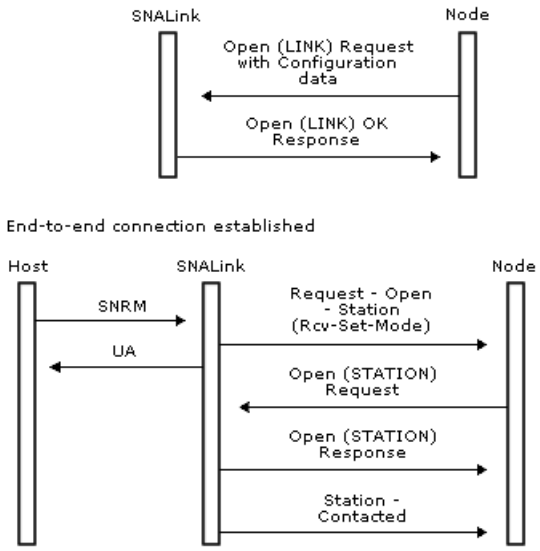
For a connection to a host computer using a leased SDLC line, the SNALink receives an SNRM when the end-to-end connection is established. The SNALink responds with a UA and informs the local node that the connection is ready for data transfer. This is done with the Request-Open-Station message with the Rcv-Set-Mode flag set.

The node then opens the STATION LPI connection with the Open(STATION) message. If the SNALink has an available control block, it responds with an Open(STATION) OK Response. This is followed by a Station-Contacted message.

A channel connection is treated the same way as a leased secondary SDLC connection. Each channel connection is associated with a channel subaddress in the range 0x00 to 0xFF. The SNA Service node will send the channel link service an Open(LINK) Request for each configured channel connection when the connection is activated. The link service should expect to receive multiple Open(LINK) Requests, one for each supported subchannel address.

Note that the Request-Open-Station message flows on the LINK LPI connection, whereas the Station-Contacted message flows on the STATION LPI connection.

The message flow for a leased line is shown in the following figure.

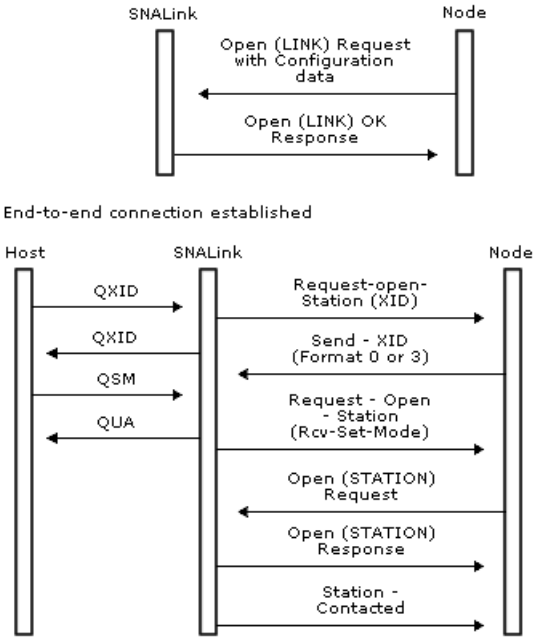


X.25, 802.2, or Switched SDLC Line (XIDs Exchanged)

The initial sequence for a host connection over X.25, 802.2, or a switched SDLC line is similar to the sequence over a leased line. The only difference is that XIDs are exchanged before the host (or front end processor) sends a mode-setting command such as QSM on an X.25 QLLC link.

When the SNALink receives an XID, it is passed to the local node on a Request-Open-Station message (on the LINK LPI connection). The local node then passes DLC a Send-XID message (also on the LINK LPI connection) containing the XID to be sent to the host. The host typically checks the node identifier in this XID and, if it is valid, sends the mode-setting command.

The sequence is shown in the following figure.



At the link level, a mode-setting command is required to enter the information transfer state. For an SDLC link, this is an SNRM (set normal response mode); for an X.25 QLLC link, this is a QSM; for a 802.2 link, it is an SABME. If the SNALink acknowledges the command (with a UA/QUA) immediately, it needs to buffer up any data that arrives on the link in the short window before the Open(STATION) Request arrives from the local node. The alternative is to wait for the Open(STATION) Request before sending the UA/QUA response.

For switched connections using SDLC modems, the Open(LINK) Request contains dial digits for manual or auto-dial modems. It is the responsibility of the SNALink to handle the management of these devices. For X.25 and 802.2 connections, the Open(LINK) Request contains the address of the remote station.

The SNALink should initiate the dialing procedure when it receives the Open(LINK) Request.

Activating a Peer Connection

For a peer connection, there is an activation sequence that involves the two stations exchanging format 3 XID frames. As part of this sequence, the two stations agree on their link roles. They also exchange information relating to the link level connection, such as the maximum frame size supported.

The node passes XIDs to the SNALink over the LINK LPI connection using the Send-XID message. The SNALink returns received XIDs to the local node over the LINK LPI connection using the Request-Open-Station message.

Fixed Link Roles and Negotiable Link Roles show examples of XID exchange for the two cases:

- The link roles are explicitly configured for the two stations.
- The link roles of both stations are negotiable.

Points to note are:

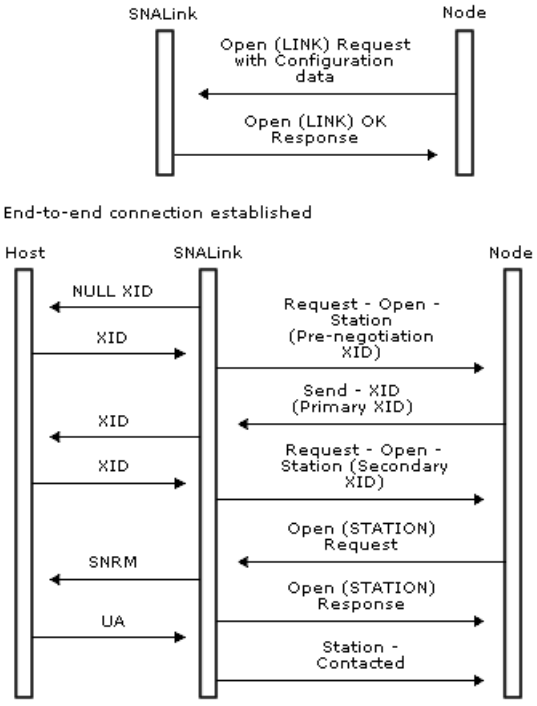
- The Open(LINK) Request is supplied with a NULL XID that is sent when the end-to-end connection is established.
- After the first NULL XID, all XIDs are format 3.
- If both stations are set up to be negotiable, the station with the higher node identifier becomes the primary.
- If both stations are negotiable and have the same node identifier, both stations produce randomized node identifiers that are compared as before.

This section contains:

- Fixed Link Roles
- Negotiable Link Roles

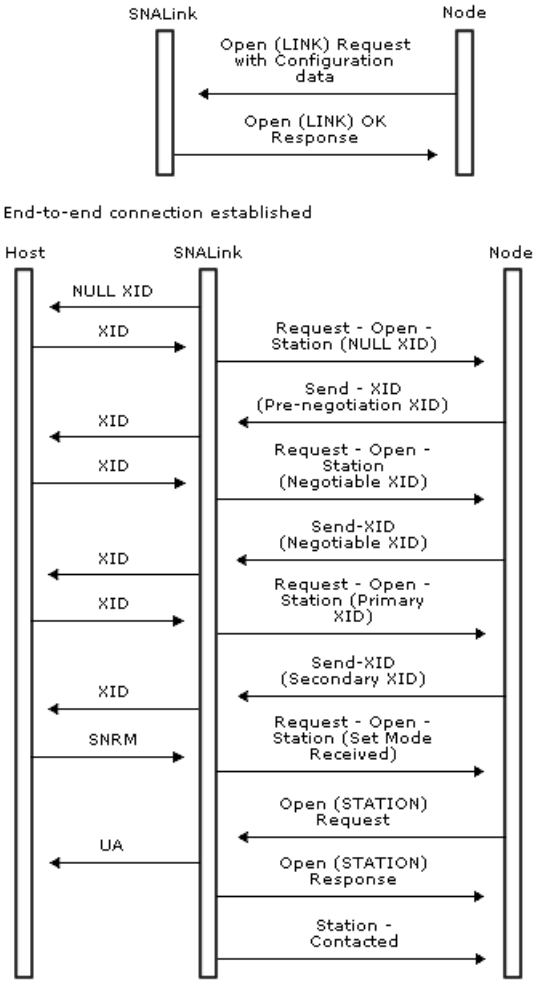
Fixed Link Roles

The following figure shows the sequence of messages for a peer connection where the local end is configured as the primary station and the remote end is configured as the secondary.



Negotiable Link Roles

The following figure shows the sequence of messages for a peer connection where both the local and remote ends are configured as negotiable. Since the remote node identifier is larger (numerically) than the local node identifier, the remote station will become primary.



local node identifier	= 0x05D11111
remote node identifier	= 0x05D22222

In summary, here are the rules that the SNALink must follow when supporting XID exchange, and in particular XID role negotiation:

- If an XID is supplied in the Open(LINK) Request, it must be transmitted as soon as the end-to-end connection is established for primary or negotiable links.
- All XIDs received from the remote station must be passed to the local node in a Request-Open-Station message.
- An XID received from the local node in a Send-XID message must be transmitted immediately.
- XID transmissions must be retried until an XID is received from the remote station. For half-duplex links, the retry time-out should be randomized to prevent repeated XID clashes.
- When a mode-setting command (SNRM, QSM, SABME) is received before the station has been opened, a Request-Open-Station must be sent to the local node with the Rcv-Set-Mode flag on.
- When the local node sends an Open(STATION) message, the link should examine it to determine its link role (that is, primary or secondary).
- A secondary station should send a Station-Contacted message after receiving and responding to the Open(STATION) message.
- For a primary station, the mode-setting command should be sent when the Open(STATION) message is received. The Station-Contacted message should be sent to the local node when this command has been acknowledged by the secondary station (for instance, a UA received on an SDLC link).

If the local node detects an error during role negotiation, such as both PUs configured as primary, it sends out an XID containing an error vector. The vector is appended to the end of the normal XID data. The vector number specified is 0x22, and the vector

data specifies that the DLC role field is in error. Refer to the *SNA Formats* Help file for details of the XID formats and error vectors.

After sending the error XID, the local node sends a Close(LINK) message to terminate the connection (see Closing a Connection).

The following figure is a matrix of the possible combinations of station link roles and shows the eventual role of the local station.

		Local Station		
		Primary	**Secondary**	**Negotiable**
Remote	**Primary**	Fail	Secondary	Secondary
Station	**Secondary**	Primary	Fail	Primary
	Negotiable	Primary	Secondary	Either*

*The station with the higher node identifier becomes the primary.

Opening the STATION LPI Connection

After receiving a Request-Open-Station (RQOS) message, the local node sends an Open(STATION) message to the SNALink when:

- The station is configured as secondary (or has negotiated to secondary), and the RQOS message has the Rcv-Set-Mode flag on, indicating that the SNALink has received a mode-setting command such as SNRM.
- The station is configured as primary (or has negotiated to primary), the RQOS message contains a secondary XID, and the local station has sent at least one negotiation-proceeding XID.

The Open(STATION) Request contains the link index that was on the Open(LINK) Request. This field is used to correlate the LINK and STATION LPI connections for SNALinks that support multiple connections, such as X.25, 802.2, and channel.

This Open(STATION) Request contains configuration data for the link station, such as the SDLC address of the adjacent link station (0x00 for secondary stations, 0x01 to 0xFE for primary stations).

The SNALink should use this address field for determining the local station's role. If it is set to 0x00, the local station is a secondary. Otherwise, the local station is primary. This field has this meaning even when the address is not used because of the link type (such as 802.2).

See Open(STATION) Request for the format of this message.

If the station cannot be initialized (perhaps due to a lack of resources), the SNALink responds with an Open(STATION) Error Response containing the appropriate error code.

Node Identification and Signaling Information

Refer to Incoming Call Support for details of the role an SNALink plays in node identification.

When XIDs are exchanged, there are two mechanisms for identifying the remote station:

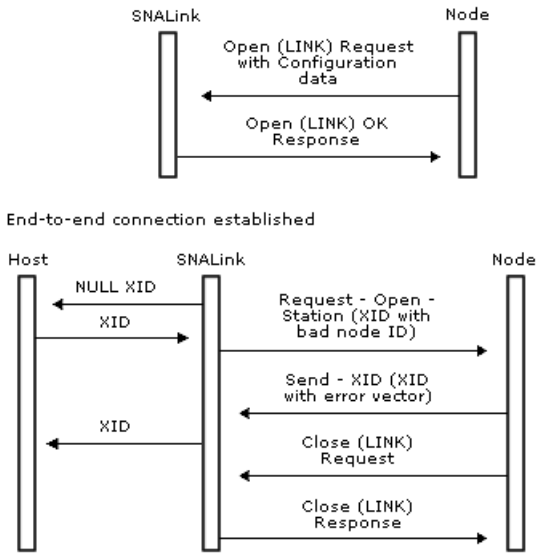
- The node identifier on received XIDs.
- The DLC defined address (for example, the MAC address). This is known as signaling information.

The presence of signaling information depends on the type of the SNALink. For instance, there is no signaling information over an SDLC link, but there is signaling information over X.25 and 802.2. The SNALink passes signaling information to the local node on the Request-Open-Station message by appending it after the XID.

If signaling information is present, the local node checks it against the configured value in the Dial-Digits record of the Host Integration Server 2000 configuration file. For incoming call support, this allows the local node to determine the connection that is to be activated. See Incoming Call Support for a fuller description of incoming calls.

If there is no signaling information, the local node compares the control point name on the received XID with the remote CP name in the configuration.

If the remote station is identified correctly, XID exchange proceeds as detailed in Activating a Peer Connection. However, if there is a mismatch, the local node sends an XID (in the Send-XID message) containing an error vector followed by a Close(LINK) Request, as shown in the following figure.



XID Retries

When the local node specifies that the SNALink is to send an XID, either by supplying it on the Open(LINK) Request or by sending it on a Send-XID message, it is the responsibility of the SNALink to perform any retries.

XIDs need to be retried because:

- The remote station has not been started yet.
- Frames may be lost on the line due to noise.

SDLC SNALinks should implement a contact time-out and a retry limit—values for these are provided on the Open(LINK) message. The time-out specifies how often the XID should be retried, and the retry limit specifies how many XIDs should be sent before abandoning the connection activation and sending an Outage message to the local node. The SNALink should stop retrying the XID when one of the following occurs:

- It receives an XID from the remote station.
- An Open(STATION) message is received from the node.
- A mode-setting command is received on the link.

Multiple Connections

For 802.2 and X.25 links, multiple connections can use the same physical link supported by a single instance of the SNALink software.

For each connection to a remote station, there is a LINK LPI connection and a STATION LPI connection (this is different from SDLC multipoint as described in SDLC Multipoint Connections). Hence, there can be multiple pairs of LPI connections between a local node and an SNALink. For each connection, the local node issues an Open(LINK) Request and, after XID exchange, an Open(STATION) Request.

The local node is configured with a maximum number of connections that can be active at any one time. In addition, each potential connection is configured with the address of the remote station. This information is required when activating a connection, and is included in the Open(LINK) Request for an outgoing connection.

DLC Information Transfer

When the local node has opened the LINK and STATION LPI connections and received a Station-Contacted message from the SNALink for a remote station, it can exchange data (using the DLC interface) with the PU and associated LUs at the remote station.

Data messages are contained within buffers. The transmission header (TH) of the SNA path information unit is contained within the buffer header. The request/response header (RH), if present, and request/response unit (RU) are contained within one or more buffer elements. The TH is contained in the buffer header for historical reasons. Typically this will be copied by the SNALink into the element before startd, to keep the entire frame in contiguous memory locations. See DLC-Data for a description of the DLC-Data message format.

Note that all data messages to a specified remote station flow on the associated DLC STATION LPI connection, and not on the controlling DLC LINK LPI connection.

This section contains:

- DLC Flow Control

DLC Flow Control

The flow of data messages at the DLC interface for each link station is flow controlled. For each direction of flow there is an initial credit of messages that can be transmitted.

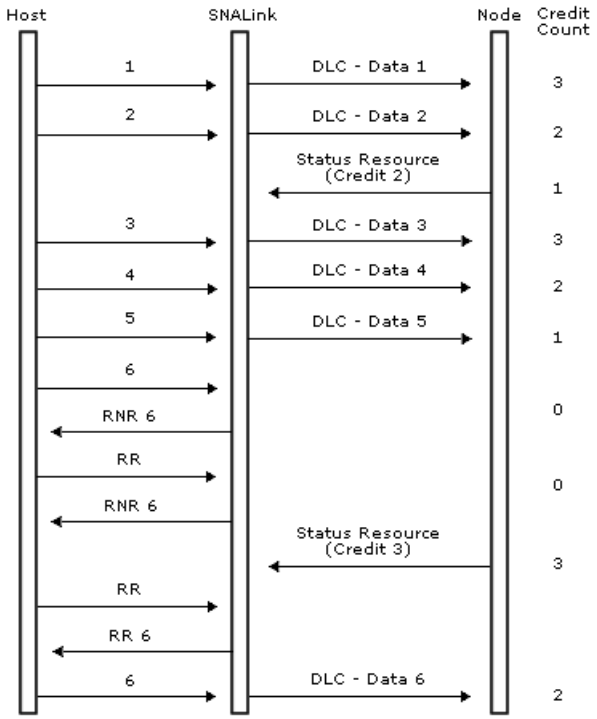
Flow control is maintained by initial specification on the Open(STATION) Request and Open(STATION) OK Response messages, and by the sending of DLC Status-Resource messages to give more credit from time to time.

The sender maintains a count of credit, starting at the initial value set on the Open(STATION), which is decremented for each DLC-Data message sent. When the credit count reaches zero, no more DLC-Data messages can be sent until more credit is received.

For flow in a given direction, the amount of credit is specified by the recipient of the data, since the recipient has to do any queueing. The initial credit values are passed on the Open(STATION) message (on the request for flow from the SNALink to the local node and on the response for flow from the local node to the SNALink).

The initial credit for the flow from the SNALink to the local node is determined by the node. The initial credit for the flow from the local node to the SNALink is set by the SNALink software—a suggested value is 16.

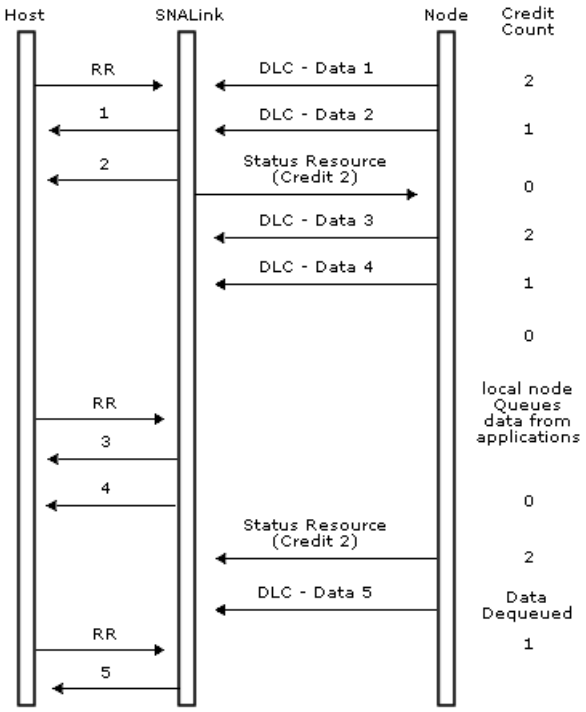
If the SNALink runs out of credit to send to the local node, it should either queue the data or discard it and send no acknowledgment. It should also start sending RNR (receive not ready), for example, when polled by a primary station. An example message flow with an SDLC SNALink is shown in the following figure with an initial credit of 3. When the SNALink runs out of credit, it does not acknowledge any further frames and starts sending RNR.



For flow control from the local node to the SNALink, when the node runs out of credit it queues the data and applies back pressure on sessions using that station. There is thus end-to-end flow control in this direction, independent of any SNA pacing that may be in force.

The SNALink gives credit to the local node for the messages that have been transmitted, not for the messages for which acknowledgments have been sent. The amount of data queueing in the SNALink is kept down most of the time since frames will usually be acknowledged.

Flow control for the flow of data from the local node to the SNALink is shown in the following figure, where the initial credit is assumed to be 2.


Host
SNALink
Node
Credit Count
RR
DLC - Data 1
2
1
DLC - Data 2
1
2
Status Resource (Credit 2)
0
DLC - Data 3
2
DLC - Data 4
1
0
RR
local node Queues data from applications
3
4
0
Status Resource (Credit 2)
2
DLC - Data 5
Data Dequeued
RR
1
5

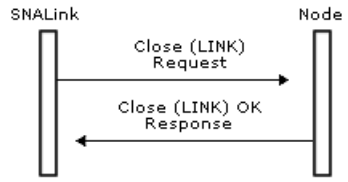
Closing a Connection

The local node closes a connection to a remote station:

- If the system administrator manually deactivates the connection.
- If the connection is configured as on-demand and no 3270 or LU6.2 sessions are active.
- If an outage has been reported by the SNALink.

The local node closes a connection by sending a Close(LINK) message. The SNALink then takes some action, such as lowering DTR (data terminal ready) on an SDLC link or issuing a DLC_CLOSE_STATION on a 802.2 connection. It then replies with a **Close(LINK) OK Response** as shown in the following figure.

The case of multipoint connections is slightly different and is considered in SDLC Multipoint Connections. The rest of this section is concerned with point-to-point connections.



This section contains:

- Outages
- Connection Retries

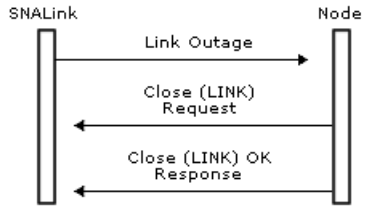
Outages

If the SNALink detects a link or station failure, it reports the failure by sending an Outage message to the node on either the LINK or STATION LPI connection depending on whether it is a link or station outage. Generally, a station outage indicates a problem at the remote station, and a link outage indicates a local or line problem.

When the local node receives an Outage message, it:

- Logs an error containing the outage code.
- Tidies up each session using the connection and informs applications of the failure (for instance, with a Comm Check code on a 3270 emulator).
- Sends a Close(LINK) Request to the SNALink.

On receipt of the **Close(LINK) Request**, the SNALink should clear up its internal resources for the connection and send back a Close(LINK) Response.



There is a special case when the node loses contact with the SNALink software. In this case the node is notified of this event (a lost locality) and performs outage processing apart from sending messages to the SNALink.

The outage codes are not distinguished by the node, but they are logged. For the sake of consistency across SNALink implementations, the values listed below should be used.

This section contains:

- SDLC Outage Codes
- 802.2 Outage Codes
- X.25 Outage Codes
- DFT Outage Codes

SDLC Outage Codes

The following table describes SDLC outage codes.

0x0D	Internally generated for SNALink lost locality.
0x11	DSR failure.
0x12	CTS failure.
0x14	DCD failure.
0x24	Nonproductive receive retry limit exceeded.
0x25	Idle time-out retry limit exceeded.
0x29	Connection problem. subqual = 0x00I-frame retransmission subqual nonzeroXID retransmission
0x2D	Abnormal modem response.
0x2E	Write time-out retry exceeded.
0xA0	XID exchange failed on multidrop line. subqual is address of secondary station.
0x15	DISC received.
0x23	Receive buffer overrun.
0x2C	Invalid command received. subqual = 0x03invalid N(R) subqual = 0x04invalid or unsupported command/response subqual = 0x05excess I-field
0x80	DM received in information transfer state.
0x81	Discontact retry limit exceeded.
0x82	Contact retry limit exceeded.
0x83	Poll retry limit exceeded.
0x84	No Response retry limit exceeded.
0x85	Remote busy retry limit exceeded.
0x86	FRMR received. subqual = 0x00no reason given subqual = 0x03invalid N(R) subqual = 0x04invalid or unsupported command/response subqual = 0x05excess I-field
0x87	Invalid frame received. subqual = 0x03invalid N(R) invalid or unsupported command/response subqual = 0x05excess I-field
0x88	RIM received.
0x89	RD received.

802.2 Outage Codes

The following table describes 802.2 outage codes.

0x29	Remote node not active.
0xAB	SABME received while connection active.
0xAC	FRMR sent.
0xAD	FRMR received.
0xAE	DISC/DM received.
0xAF	Link lost.

X.25 Outage Codes

The following table describes X.25 outage codes.

0x37	Loss of a virtual circuit.
0x60	SVC cleared down by remote station or network.
0x61	PVC has been reset by remote station or network.
0x62	Attempt to connect to remote station through SVC failed.

DFT Outage Codes

The following table describes DFT outage codes.

0x11	DSR failure.
0x14	DCD failure.
0x15	Connection terminated by host.

Connection Retries

If an initially active or operator-started connection is closed because of an outage, the node periodically tries to reopen the connection. This can be stopped by manually deactivating the connection. This retry mechanism is also used when the node attempts to open a connection but the SNALink software has not yet been started. On-demand connections are not retried automatically by the node, but will be retried if the user attempts to reactivate a session using the connection.

Note that this connection retry timer is totally separate from the timers specified on the Open(LINK) Request.

Incoming Call Support

The local node allows an SNALink to be set up to support incoming calls. In this mode of operation, the node primes the SNALink by sending an Open(LINK) Request, but the SNALink does not attempt to activate the link until it receives an XID from a remote station.

The SNALink recognizes an Open(LINK) Request for an incoming call by the absence of a connection name in the destination name field (this field is filled with ASCII blanks).

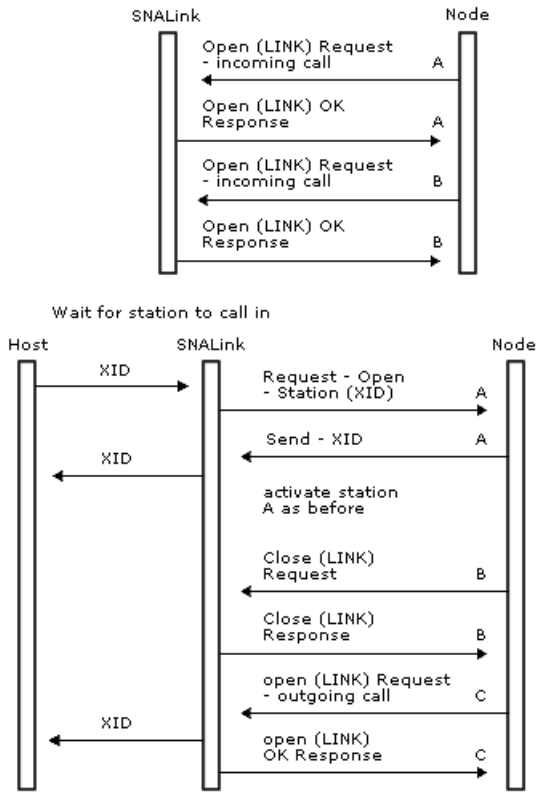
For incoming calls, Open(LINK) Request requires an immediate response from the SNALink, just as in the case of an Open(LINK) Request for an outgoing call.

For an SDLC SNALink there can be only one Open(LINK) outstanding. However, 802.2 and X.25 allow the possibility of multiple connections being handled through a single SNALink. In these cases, for each configured connection that is primed to await incoming calls, the local node will send an Open(LINK) with a blank connection name to the SNALink.

When an incoming call is received by the SNALink, the received XID should be passed to the local node on any LPI connection that is primed for incoming calls. The LPI connection selected must then be used for all future messages relating to that incoming call.

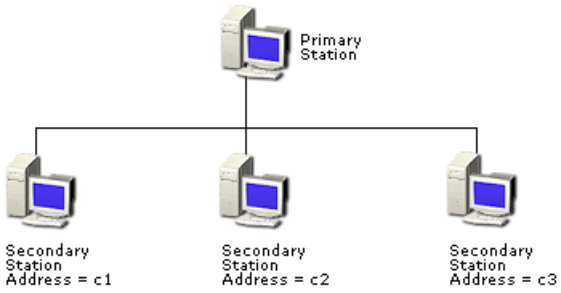
It is not necessary for the SNALink to perform validation of incoming calls—this will be performed by the local node. However, if required, the SNALink can choose to validate calls before passing them to the node. A common example of this is to ensure that only X.25 calls with a specific local address are passed through to the local node.

The following illustration shows incoming call support with an SNALink that supports two connections. A remote station calls in and uses connection A. The node sets up connection B for incoming calls but then needs to open connection C. Since the SNALink only supports two connections, connection B is closed.



SDLC Multipoint Connections

The node can support primary multipoint (also known as multidropped) links, at both the primary and secondary end. Multipoint is a special configuration for an SDLC leased link where a single SDLC line at the primary station can be used to communicate with up to 16 secondary stations. Special hardware is required to fan out the primary line so that there is a physical connection to each secondary station. The following figure shows an example with three secondary stations.



The SDLC address of the secondary station is used to route frames to and from the individual secondary stations. Hence, the SNALink at the secondary station needs to check the SDLC address as the primary sends all frames to all secondary stations. The SNALink at the secondary station should only accept frames with its SDLC address—the other frames should be ignored.

From the viewpoint of the node at a secondary station, the message flow at the DLC interface is as for a point-to-point connection (described in Opening a Connection). Indeed, the node need have no knowledge that this is a multipoint connection.

The primary end has to handle the special processing required for multipoint connections. The remainder of this section concentrates on the primary station.

At the primary end, there are the following LPI connections:

- One LINK LPI connection.
- A STATION LPI connection for each active secondary station.

Since the XID exchange is carried out using the single LINK LPI connection, the Request-Open-Station and Send-XID messages always specify the station address of the secondary station that the XID has arrived from or is going to. Note that no XID is supplied on the Open(LINK) Request.

Each STATION LPI connection has different values of I, the index. After the station has been activated, data messages flow on the STATION LPI connection rather than the LINK LPI connection.

The first figure below shows the activation of two stations on a multipoint link, from the viewpoint of the primary end. The full exchange of format 3 XIDs for each station is not shown but is the same as in Opening a Connection. In addition, the figure shows the stations being activated one after the other. In fact, the stations can be activated simultaneously.

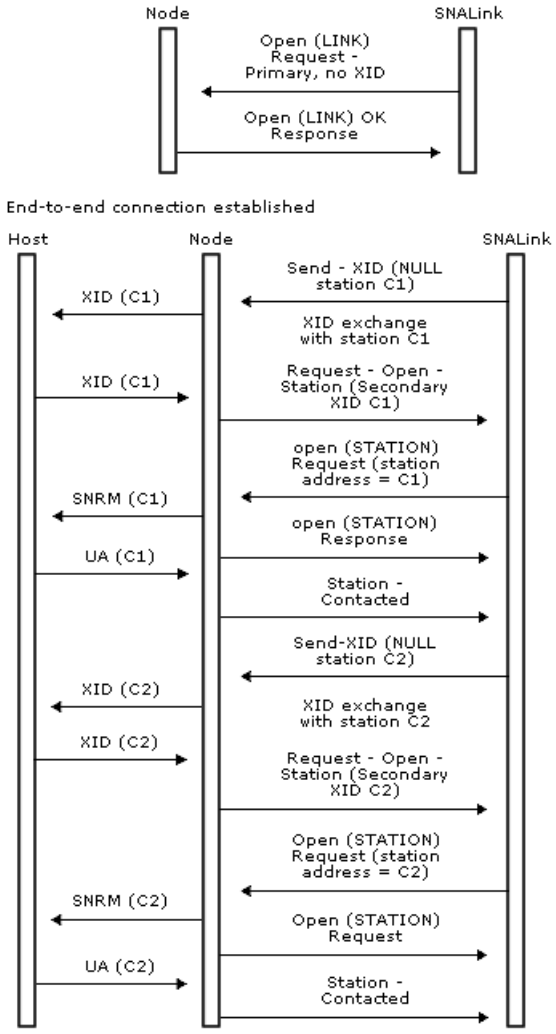
If the XID exchange fails because the secondary is failing to reply to the XID, the SNALink generates a special variant of link Outage message. Ideally, the SNALink would give a station Outage message, but this is not possible because the STATION LPI connection is not yet open. Instead, the SNALink generates a link Outage message with code 0xA0 and a subqualifier that is the SDLC address of the station.

When the stations are activated on a multipoint link, the majority of messages flow across the STATION LPI connections. If a connection to a particular secondary station is to be closed (because the operator deactivates it, for instance), the node issues a Close(STATION) Request. The SNALink replies with a Close(STATION) Response to the node and sends a DISC frame to the secondary station.

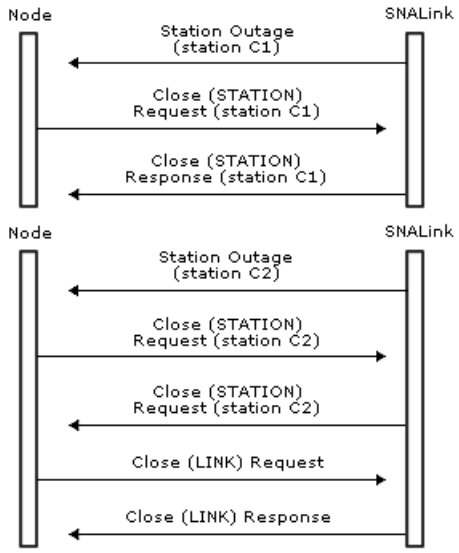
The SNALink can generate both station and link Outage messages. If the problem only affects a particular station, such as not responding to polls, the link generates a station Outage message and the node closes the station with a Close(STATION) Request. The SNALink responds with a Close(STATION) Response.

If the problem affects the link as a whole, such as the line being disconnected from the primary SDLC adapter, the SNALink generates a link Outage message and the node sends a Close(LINK) Request. The SNALink responds with a Close(LINK) Response.

Whenever the node receives a Close(STATION) Response, it checks to see if any stations are still active on the multipoint link. If not, a Close(LINK) Request is sent. The SNALink responds with a Close(LINK) Response. The following figure shows the message flows for outage processing. It shows a multipoint connection with two secondary stations (the full XID exchange is not shown).



The following illustration shows close processing for a multipoint configuration with two secondary stations. Outage conditions are detected on both stations.



Note that the station messages are labeled in the figure with station addresses. In fact, the node and SNALink use the LPI addresses to identify the two stations.

Setup Information

This section describes the integrated link service installation provided with Microsoft® Host Integration Server 2000 and the earlier procedures used with Microsoft® SNA Server 3.0 and Microsoft® SNA Server 4.0. This section also describes the older Microsoft® Windows NT® INF-based link service installation procedures used in SNA Server 2.*x*.

In Host Integration Server 2000, SNA Manager is used to install and configure link services. Host Integration Server 2000 uses the Microsoft System Installer (MSI) and MSI packages for the installation of the Host Integration Server software. Link services from Independent Hardware Vendors (IHVs) are not included in the main Host Integration Server MSI packages. IHV link services are installed using a separate IHV-provided MSI package. A sample IHV link service using the Generic SDLC link service that illustrates IHV link service installation is included in the Host Integration Server Software Development Kit (SDK). The SDK samples are installed on your computer when the SDK option is selected during installation of Host Integration Server software. These sample files are also located on the Host Integration Server 2000 CD-ROM under the SDK\Samples\IHVLInks subdirectory. Host Integration Server 2000 does not support the earlier INF-based link service setup procedure.

In SNA Server 3.0 and SNA Server 4.0, SNA Explorer is used to install and configure link services. SNA Server 3.0 and SNA Server 4.0 support the INF-based link service setup procedure, but there are clear benefits in updating your link service to the integrated link service installation procedure introduced in SNA Server 3.0.

This section contains:

Setup Registry Architecture

There are two main subtrees in the Microsoft® Windows® 2000 and Microsoft® Windows NT® registry where information is kept relevant to Microsoft® Host Integration Server 2000 and Microsoft® SNA Server: the **SOFTWARE** tree and the **SYSTEM** tree. Both of these are subtrees of **HKEY_LOCAL_MACHINE**. The **SOFTWARE** tree contains generic information about IHV link services, and the **SYSTEM** tree contains information about the individual components of those services. While reading the following topics, it may be helpful to view examples of what is being discussed by inspecting the registry of an existing system with several of the built-in link services installed.

This section contains:

Product Entries

All of the information relevant to the product as a whole resides in the registry under the key **SOFTWARE\Microsoft**. Each product or link support has an entry whose name consists of the product name and version separated by an underscore. This key contains most of the information about the product, such as the script name and option name that control it and the service name for that particular instance.

Each instance key must also have a NetRules key. This key contains all of the information for the Network Control Panel Applet bindings.

The Host Integration Server and SNA Server setup writes the path of the root directory of the machine's Host Integration Server or SNA Server tree into the key:

SOFTWARE\Microsoft\SNA Server\CurrentVersion\Setup\RootDir.

Service Entries

Each instance of a component appears to the system as a unique service. These services must be created using the Service Control Manager (SCM). The SCM creates a registry entry for each service under **SYSTEM\CurrentControlSet\Services**. This key contains all of the service-specific information.

The top-level service key contains information that the SCM uses to control the service. This includes the type of service this key represents, how it should be started, what sort of error handling should be used, the path to the executable image, and so on. All information in this key should be handled by the SCM. Each service key also contains two subkeys: the Linkage key and the Parameters key.

The Linkage key is used by the Network Control Panel Applet to store binding information. The Parameters key contains information that is relevant to SNA Server Setup, such as the name of the DLL responsible for handling a link service. All information in this key should be handled by SNA Server Setup. The Parameters key contains another key, ExtraParameters, which is used for any IHV-specific information, including component-specific parameters and other information not required by the main SNA Server setup program.

Integrated Link Service Setup on Host Integration Server

In Microsoft® Host Integration Server 2000, the SNA Manager supports installation and configuration of link services. Host Integration Server 2000 uses the Microsoft System Installer (MSI) and MSI packages for the installation of the Host Integration Server software. Link services from Independent Hardware Vendors (IHVs) are not included in the main Host Integration Server MSI packages. IHV link services are installed using a separate IHV-provided MSI package. For an example of this process, install the DLC/802.2 or IBM SDLC link service in Host Integration Server.

IHV MSI Packages contain two types of features:

- Features that can be installed and used independently of Microsoft Host Integration Server 2000
- Features that require Microsoft Host Integration Server 2000 to function.

Features that can be installed and used independently of Microsoft Host Integration Server 2000 include drivers, utilities and applications that can run without Host Integration Server support. These features should be represented in the package as one or more features in the MSI Select Features dialog. (refer to Generic Link Service sample feature “Generic Link Service” from the Generic.msi Feature table)

Features that require Microsoft Host Integration Server 2000 include drivers, utilities and applications that require Host Integration Server to function. These features should be represented in the package as one or more features in the MSI Select Features dialog. These features should be hidden if Host Integration Server is not installed on the computer. (refer to Generic Link Service sample feature “Host Integration Server Support” from the Generic.msi feature table)

Properties can be equated to a variable (either global or local) in a high level programming language such as C or C++. Properties can be used as a placeholder for informational text, or as values used during an installation. (refer to property SERVER_INSTALLED in the custom action source code GenSet.cpp, and the Condition table entry in the Generic.msi package).

Custom Actions provide a method of extending the capabilities of MSI. Functions not supported in MSI, can be custom written, and invoked from within a sequence table or directly from a dialog control event. (refer to Custom Actions SetHISPath and GetHISData in the GenSet.cpp source file)

Launch Conditions provide a method of preventing an install from launching. The sample MSI package included with the Host Integration Server SDK doesn’t use a launch condition, however if your package requires Host Integration Server to be installed for your features to function, you should include a launch condition that fails the installation if Host Integration Server is not detected)

The Condition table provides a method of controlling the layout of the features listed in the select feature dialog. The sample MSI package uses the Condition table to hide the “Host Integration Server Support” if Host Integration Server is not detected in the installation..

In order for Host Integration Server to control the installed state of IHV features that require Host Integration Server to be installed, the following registry keys must be included in the package for each separate feature which requires Host Integration Server to be installed.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SnaBase\IHVSupport\<ihv key>
<feature table entry 1> : REG_SZ : <product code 1>
<feature table entry 2> : REG_SZ : <product code 2>
.
.
.
<feature table entry n> : REG_SZ : <product code n>
```

where:

<feature table entry x> - specifies an entry in the feature table (not title)

<product code x> - specifies the package product code

Note that feature table entries must be unique. Product codes may or may not be unique depending on the number of packages provided by an IHV. One package may have several Host Integration Server dependent features and therefore should list each feature separately.

These registry keys should be installed by the feature rather than globally by the package.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SnaBase\LinkServicesInstalled
<Link Service Name> : REG_SZ : <Link Service Configuration Dll name>
```

Target Paths specify a directory where files will be installed. An MSI package will contain one or more Target Paths.

The sample package contains two features:

- Generic Link Service
- Host Integration Server Support

The Generic Link Service is not dependent on Host Integration Server. This feature installs the following:

- gencfg.exe into the <Generic Link Service> directory.
- generic.sys driver into the %windir%\system32\drivers directory.
- generic.inf into the %windir%\inf directory

Host Integration Server Support is dependent on Host Integration Server. This feature installs the following:

- gendtct.dll into the HIS\system directory.
- generic.dll into the HIS\system\hwsetup\i386 directory.

This feature contains the following entry in the condition table to hide the feature if Host Integration Server is not detected

```
HIS_RELATED_FEATURE  0  SERVER_INSTALLED=”NO”
```

See dialog snapshots below for layout of features with/without HIS installed:

Adds:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SnaBase\LinkServicesInstalled
Generic Link Service : REG_SZ : GENERIC.DLL

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SnaBase\IHVSupport\GenericLinkService
HIS_RELATED_FEATURE : REG_SZ : {FDF11E0E-3BFF-4B0F-89BD-E4E1FB979E4D}
```

The sample package uses one custom action dll with two entry points:

```
SetHISPath
GetHISData
```

The **SetHISPath** entry sets the target path to the Host Integration Server 2000 install directory.

The GetHISData entry sets the MSI Property SERVER_INSTALLED to “YES” if Host Integration Server is installed and sets the MSI Property SERVER_INSTALLED to “NO” if Host Integration Server is not installed.

The sample contains two Target Paths:

```
INSTALLDIR
INSTALLDIR1
```

The INSTALLDIR target path specifies the installation directory for the non Host Integration Server dependent features. It can be set using the “browse” button in the “Select Features” dialog when the “Generic Link Service” feature is currently selected.

The INSTALLDIR1 target path specifies the directory where Host Integration Server is installed. This Target Path is set by the custom action “SetHISPath”.

The IHVUtil.exe tool can be used to verify the Host Integration Server dependent interfaces. If the tool is launched after the IHV package is installed, all Host Integration Server dependent features should show up in the dialog. Executing “Remove” from the dialog should remove the Host Integration Server dependent feature as well as remove it from the dialog.

Note that the sample provided can also be tested using the SNA Manager. While the sample does not actually function, it will appear as an installed link service.

This section contains:

- Integrated Link Service Configuration and Reconfiguration on Host Integration Server

- Constructing an Integrated Link Service DLL on Host Integration Server

Integrated Link Service Configuration and Reconfiguration on Host Integration Server

In Microsoft® Host Integration Server 2000, the initial configuration functions are performed by your configuration DLL running in the context of SNA Manager.

After your link service has been created, SNA Manager must be able to locate its configuration DLL when the operator wishes to reconfigure the link service. To support this feature, when your configuration DLL initially creates the link service, it must put a new value in the registry of the target server as follows:

SYSTEM\CurrentControlSet\Services\<*yourLinkService*>\Parameters

DLLName: REG_SZ: <*configDllName*>

where:

<*configDllName*> is the file name and extension of the configuration DLL, for example, IBMSDCFG.DLL. No path is specified in the value.

This value replaces InfName, which was used in SNA Server 2.*x* to name the path to the .INF file.

Since SNA Manager can be running on a management workstation remote from the target server, the configuration DLL must be able to create configuration information on the target server. The sample configuration DLL included with the earlier SNA Server SDK sample files illustrates the utility functions available to perform these functions across a network connection on the target server. Host Integration Server loads the appropriate configuration DLL over the network from \<snaRoot>\SYSTEM\HWSETUP\<cpu> on the target server as needed.

> **Note** There is an alternate way of locating the Link Service Configuration DLL if the link services from the vendor were not included with the released Host Integration Server CD-ROM. Depending on the setup tool used by the vendor, the vendor's setup software may not be able to read the registry and locate the directory where link services should be installed. To resolve this problem, the SNA Manager scans the **LinkServicesInstalled** key prior to making the call to the Link Service Configuration DLL. The SNA Manager checks for a % character in the configDllName and if it exists configDllName will be interpreted differently than just the name of the configuration DLL. The following example illustrates this case:
>
> Under the SYSTEM\CurrentControlSet\Services\<yourLinkService>\Parameters key
>
> DLLName: REG_SZ: "share\%s\<relative path and DLL Name>
>
> If a %s string is found, then \\ServerName will be prepended and the CPU architecture string (i386 or Alpha) will be substituted for %s.

Constructing an Integrated Link Service DLL on Host Integration Server

Microsoft® Host Integration Server 2000 provides an enhanced method for installing integrated link services that allows for remote setup and administration of new link services, as well as support for setup and configuration using a command-line tool. This feature is based on the link service provider supplying a setup, and configuration DLL exporting a specific list of functions. A developer must follow certain standards for using this SNA link service configuration DLL and set various keys and values as registry settings to be used by Host Integration Server for link service configuration.

In order to support vendors using these setup and configuration features, Host Integration Server includes the source code for a sample generic SDLC link service configuration DLL (linkcfg). Also included for use by developers is the source code to a library of utility functions (lnktools) that are commonly useful when implementing the linkcfg DLL. This sample code and the documentation that follows can be used as a guideline for vendors developing similar link service configuration DLLs for their hardware. This sample source code is located on the Host Integration Server CD-ROM in the \SDK\SAMPLES\IHVLinks subdirectory. Sample include files, C++ files, resource files, makefiles, and project files are included for use with Microsoft® Visual C++® version 6.0 and later.

This section contains:

Components of an Integrated Link Service Configuration DLL on Host Integration Server

The link service configuration DLL must export the following functions:

Exported function	Purpose
CommandLineAdd	Called from **LinkCfg** to parse command-line input.
ConfigureLinkService	Called from SNA Manager to add or modify a link service.
ConfigureLinkServiceEx	Called from SNA Manager to add or modify a link service, returning a configuration buffer to be added to the configuration file.
DisplayHelpInfo	Return a buffer containing command-line syntax for this type of link service.
RemoveLinkService	Called from SNA Manager to remove a link service.
RemoveAllLinkServices	Called from Setup to remove all instances of this link service.

The sample linkcfg.cpp DLL is written in C++ using the Microsoft Foundation Classes (MFC) and uses a single property sheet with two property pages as follows:

- The card configuraton property page implementation is in the cardcfg.cpp and cardcfg.h files. This property page is concerned with configuring various hardware properties (interrupt, DMA channel, and I/O address, for example) of the link service hardware.
- The connection mode property sheet implementation is in the mode.cpp and mode.h files. This property page is concerned with configuring mode information (link service name, link service title, SDLC line type, for example) for the link service.

The two property pages are linked to the link service property sheet in linkcfg.cpp within the **ConfigureLS** routine. This function is called by the exported **ConfigureLinkService** and **ConfigureLinkServiceEx** routines in linkcfg.cpp. An actual link service configuration DLL developed from these sources may require more property pages depending on the information needed to configure the actual link service DLL.

The registry.h include file used by linkcfg.cpp contains a global definition of the registry entries required for the sample generic SDLC link service. The values in this structure will be modified to contain the actual information specified by the user. This structure is added to the registry when a new link service is configured, and this structure is removed when a link service is deleted. The registry values that a developer must modify include the Link Registry Base entry (LINKSERVICE is used in the sample include file), the name of the device driver root (GenSdlc is used in the sample include file and source code), and various software and service registry settings appropriate for the target link service.

Several of the exported link service DLL functions use a configuration buffer, the CONFIG_BUFFER structure defined in linkcfg.h. The format of any CONFIG_BUFFER used by developers must match the structure format of this sample file for the first three parameters. Other parameters may differ for a developer's version of the CONFIG_BUFFER structure based on the target link service.

The sample link service configuration DLL calls a set of general utility functions that are not specific to any target link service. These utility functions are included in a lnktools library (lnktool.cpp) that is linked in as an OBJ file. This lnktools library includes the following utility functions that are useful in developing link service configuration DLLs:

Utility function	Purpose
AddPerfmonCounters	Add perfmon counters for this link service.
bCreateService	Create a service on a computer.
bDeleteService	Delete a service on a computer.
bStopService	Stop a service running on a computer.
CheckForExistingLinkService	Check to see if a link service of this type exists with this title.
ConvertHexStringToDWORD	Convert a hexadecimal string to a DWORD value.
ExtractNextParameter	Get the next parameter from a buffer.
fAddRegistryEntry	Add a new registry value to the registry.
fCanWeAdministerRemoteBox	Determine if the user has administrative privileges on the remote computer.
fConnectRegistry	Connect to a remote computer's registry and return a handle to the remote registry.
fDisconnectRegistry	Disconnect from a remote computer's registry.
fFindAndReplaceString	Find and replace a substring within a string.
fFindString	Determine if a string exists within a string buffer.
fFindStringInMultiSZ	Find a string in a REG_MULTI_SZ string list and return entire string.

fQueryRegistryValue	Query a value from the registry.
fRegistryKeyExists	Test whether a registry key exists.
fRemoveRegistryEntry	Remove a registry key.
fRemoveRegistryValue	Remove a registry value.
fStringCompare	Determine if two strings compare.
LoadStringResource	Load a string from the string resource.
ParseNextField	Return the next field from a string.
RemovePerfmonCounters	Remove perfmon counters for this link service.
ReturnString	Return a pointer to a string resource string.

The sample source code for a generic SDLC link service configuration DLL (linkcfg) includes several functions that may be useful as sample code when developing link service configuration DLLs for other hardware. The following functions are included in the linkcfg.cpp source code that may be of use as examples:

Utility function	Purpose
bDetectNetworkCard	Detect the remote network card and return the card settings buffer for the sample generic SDLC link service.
bLastGenericDFTLinkService	Check for the last generic SDLC link service for the sample generic SDLC link service. This routine is used to determine if the GENSDLC Device Driver (if one existed) can be removed.
ConfigureLS	The common link service configuration function used by the sample generic SDLC link service.
fAddAllRegistryValues	Add all registry values for the sample generic SDLC link service.
fAddClassAndBindformRegistries	Add the "class" and "bindform" registry entries for the sample generic SDLC link service. The bindform and class registry entries can only exist for the first link service of this type.
fEnumerateEventLogSources	Enumerate the Event Log sources registry value for the sample generic SDLC link service.
fRemoveAllRegistryValues	Remove all registry values for the sample generic SDLC link service.
fReplaceAllRegistryValues	Replace all user-provided information in the registry data for the sample generic SDLC link service.
fReplaceRegistryData	Replace global registry data for the sample generic SDLC link service.
fReplaceRegistryKeyName	Replace global registry structure strings for the registry key name for the sample generic SDLC link service.
fSetupGlobalValues	Create or update all user-provided information in the registry data structure for the sample generic SDLC link service.
InitializeGlobalStructure	Initialize link service data contained in the global data structure for the sample generic SDLC link service.

Contents of IHVLinks Sample Kit on Host Integration Server

The sample source code for a generic SDLC integrated link configuration MSI package that illustrates an integrated IHV link service installation are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\IHVLinks subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the SDK\Samples\IHVLinks subdirectory below where the Host Integration Server SDK software is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\IHVLinks subdirectory below where the MSDN Platform SDK has been installed.

These sample files include the following directories:

Directory	Description
LinkServ	Directories used in creating the IHVLinks sample generic SDLC link service configuration and detection DLLs.
LinkServ\Build	A file containing teh normal COFF base address for a link service configuration DLL.
LinkServ\Detect	The source code to the sample generic SDLC link service detection DLL.
LinkServ\Linkcfg	The source code to the sample generic SDLC link service configuration DLL. The link service configuration DLL must export specific functions. A DEF file must be used so that exported function names are not decorated by the compiler and linker.
LinkServ\LnkTools	The source code to a collection of library routines used by the generic SDLC link service configuration DLL.
LinkServ\NT5INF	An INF file that can be used with the generic SDLC link service.
Setup	Directories used in creating the IHVLinks sample generic SDLC link service setup and setup test tools.
Setup\Bins	The compiled generic MSI package containing the generic SDLC link service driver, configuration and detection DLLs, and configuration tool.
Setup\CASource	This directory contains the source code used for the custom actions in setup. The GetHISData custom action sets the MSI property SERVER_INSTALLED to "YES" if HIS2000 is installed, otherwise the property is set to "NO". This custom action is used to disable Host Integration Server 2000 dependent features from the installation directory if not installed. The SetHISPath custom action sets the target directory INSTALLDIR1 to the installation directory where Host Integration Server 2000 was installed. This custom action is used to set the destination directory for the link service configuration DLLs.
Setup\Package	The sample GENERIC.MSI SDLC link service package ready for installation and testing.
Setup\Tools	The source code for various utility functions and tools that can be used to test your integrated link service MSI package.

This sample source code is included for your reference. The communication between workstation management station and Host Integration Server is performed by RPCSVC.EXE, the SNA RPC service.

Integrated Link Service Setup on SNA Server

In Microsoft® SNA Server 3.0 and Microsoft® SNA Server 4.0, SNA Explorer supports installation and configuration of link services. For an example of this process, install the DLC/802.2 or IBM SDLC link service. On SNA Server 3.0 and SNA Server 4.0, the benefits of updating your link service to use the integrated link service installation procedure are:

- Setup using the integrated link service provides full support for remote installation or reconfiguration of the link service.

  The link service retains access to physical hardware through an optional link service-provided helper DLL running on the target server, to support application-specific adapter detection or diagnostic functions.
- It is integrated with the look and feel of Host Integration Server Manager or SNA Explorer, for greater ease of use.
- It eliminates the older Microsoft® Windows NT® Setup Engine and .INF files.

This section updates the setup information for integrated link services, providing the information you need to develop an integrated link service installation and configuration procedure.

This section contains:

Changes from INF-Based Setup

In Microsoft® SNA Server 2.*x*, the INF-based link service setup performed both the initial setup and the reconfiguration process for the link service. In SNA Server 3.0 and later, these functions have been split apart. The product setup routine is responsible only for providing files for the link service and creating one registry entry. The remainder of the link service installation and configuration is performed in SNA Explorer.

An integrated link service must provide the following components:

1. Link service DLL, optional device driver file(s) that run on the target server. These components are essentially unchanged from SNA Server 2.x.
2. Detection DLL that also runs on the target server and performs application-specific tasks in conjunction with the configuration DLL. This detection DLL can perform any adapter maintenance function, not just hardware detection.
3. Configuration DLL (with any associated help files) that runs on the management workstation in the context of SNA Explorer. The configuration DLL exposes the configuration interface for the link service and drives functions in the detection DLL. SNA Explorer loads this DLL across the network from the target server as needed. This DLL runs on the management machine under SNA Explorer and is responsible for the configuration dialog box and property sheets for your driver and link service.

The integrated link service configuration procedure does not require changes to your link service process or device driver. The structure of registry entries used by your link service and driver needs to change in the areas noted in the following two topics.

Integrated Link Service Setup Procedure on SNA Server

For each link service selected by the operator, SNA Server setup performs these operations. If you provide a custom IHV link service setup, it must do likewise.

1. Copy the link service, device driver, and any detection DLL to the target server's \<snaRoot>\SYSTEM directory. These files do not need to be copied to \<snaRoot>\SYSTEM\HWSETUP, as was done in SNA Server 2.*x*.
2. Copy the Host Integration Server Manager configuration DLL (if any) or the SNA Explorer configuration DLL (if any) to \<snaRoot>\SYSTEM\HWSETUP\<cpu>. Since the configuration DLL may be run from a management workstation of any CPU type, you should provide multiple flavors of the configuration DLL, even if your link service itself only supports a single CPU type.
3. Add a registry entry to register your link service so that SNA Explorer can list your link service in the **Insert New Link Service** dialog box. This registry entry should be added to the following subkey:

   SYSTEM\CurrentControlSet\Services\SnaBase\LinkServicesInstalled

   This is a new subkey for SNA Server 3.0 and SNA Server 4.0 that contains one value for each link service chosen to be copied to the target server's hard disk during SNA setup. The value name and value data are as follows:

   <*value*>: REG_SZ: <*configDllName*>

   where:

   <*value*> is the friendly name of your link service, for example, "IBM SDLC Link Service".

   <*configDllName*> is the file name and extension of the configuration DLL, for example, IBMSDCFG.DLL.

   > **Note** This registry entry is defined in the registry on the target server, not on the management workstation on which Host Integration Server Manager or SNA Explorer happens to be invoked.

   Setup for SNA Server 3.0 and SNA Server 4.0 asks for a list of link services to install, and copies only those link services to the target server. This speeds up setup and reduces the disk space required for Host Integration Server. Setup does not define a Windows NT service or other registry entries as was done in SNA Server 2.*x*; it merely copies files.

   There is an alternate way of calling the Link Service Configuration DLL if the link services from the vendor were not included with the released SNA Server CD-ROM. Depending on the setup tool used by the vendor, the vendor's setup software may not be able to read the registry and locate the directory where link services are installed. To resolve this problem, the SNA Manager scans the LinkServicesInstalled key prior to making the call to the Link Service Configuration DLL. The SNA Manager checks for a % character in the configDllName and if it exists \\ServerName will be prepended and the %s will contain the processor architecture (i386 or Alpha). The following example illustrates this:

   Under the SYSTEM\CurrentControlSet\Services\SnaBase\LinkServicesInstalled key,

   "Our Link Service Friendly Name" with a REG_SZ for the configDllName as follows:

   "share\%s\<RelativePath and DLL Name>"

Integrated Link Service Configuration and Reconfiguration on SNA Server

In Microsoft® SNA Server 2.*x*, the .INF script created a Windows NT service to run your link service DLL, installed the device driver, and prompted the user for initial configuration information, in addition to supporting subsequent reconfiguration. In SNA Server 3.0 and later, the initial configuration functions are performed by your configuration DLL running in the context of SNA Explorer.

After your link service has been created, SNA Explorer must be able to locate its configuration DLL when the operator wishes to reconfigure the link service. To support this feature, when your configuration DLL initially creates the link service, it must put a new value in the registry of the target server as follows:

SYSTEM\CurrentControlSet\Services\<*yourLinkService*>\Parameters

DLLName: REG_SZ: <*configDllName*>

where:

<*configDllName*> is the file name and extension of the configuration DLL, for example, IBMSDCFG.DLL. No path is specified in the value.

This value replaces InfName, which was used in SNA Server 2.*x* to name the path to the .INF file.

Since SNA Explorer can be running on a management workstation remote from the target server, the configuration DLL must be able to create configuration information on the target server. The sample configuration DLL included with the SNA SDK sample files illustrates the utility functions available to perform these functions across a network connection on the target server. Also, since the management workstation may be of any CPU type supported by Windows NT, you must provide versions of the configuration DLL for each CPU type. SNA Explorer loads the appropriate CPU-type version of the configuration DLL over the network from \<snaRoot>\SYSTEM\HWSETUP\<cpu> on the target server as needed.

> **Note** There is an alternate way of locating the Link Service Configuration DLL if the link services from the vendor were not included with the released SNA Server CD-ROM. Depending on the setup tool used by the vendor, the vendor's setup software may not be able to read the registry and locate the directory where link services should be installed. To resolve this problem, the SNA Manager scans the **LinkServicesInstalled** key prior to making the call to the Link Service Configuration DLL. The SNA Manager checks for a % character in the configDllName and if it exists configDllName will be interpreted differently than just the name of the configuration DLL. The following example illustrates this case:
>
> Under the SYSTEM\CurrentControlSet\Services\<yourLinkService>\Parameters key
>
> DLLName: REG_SZ: "share\%s\<relative path and DLL Name>
>
> If a %s string is found, then \\ServerName will be prepended and the CPU architecture string (i386 or Alpha) will be substituted for %s.

Constructing an Integrated Link Service DLL on SNA Server

Microsoft® SNA Server version 4.0 provides an enhanced method for installing integrated link services that allows for remote setup and administration of new link services, as well as support for setup and configuration using a command-line tool. This feature is based on the link service provider supplying a setup, and configuration DLL exporting a specific list of functions. A developer must follow certain standards for using this SNA link service configuration DLL and set various keys and values as registry settings to be used by SNA Server for link service configuration.

In order to support vendors using the new setup and configuration features, SNA Server 4.0 includes the source code for a sample generic SDLC link service configuration DLL (linkcfg). Also included for use by developers is the source code to a library of utility functions (lnktools) that are commonly useful when implementing the linkcfg DLL. This sample code and the documentation that follows can be used as a guideline for vendors developing similar link service configuration DLLs for their hardware. This sample source code is located on the SNA Server CD-ROM in the \SDK\SAMPLES\LINKSERV\GENERIC subdirectory. Sample include files, C++ files, resource files, makefiles, and project files are included for use with Microsoft® Visual C++® version 4.2 and later.

This section contains:

Components of an Integrated Link Service Configuration DLL on SNA Server

The link service configuration DLL must export the following functions:

Exported function	Purpose
CommandLineAdd	Called from **LinkCfg** to parse command-line input.
ConfigureLinkService	Called from SNA Manager to add or modify a link service.
ConfigureLinkServiceEx	Called from SNA Manager to add or modify a link service, returning a configuration buffer to be added to the configuration file.
DisplayHelpInfo	Return a buffer containing command-line syntax for this type of link service.
RemoveLinkService	Called from SNA Manager to remove a link service.
RemoveAllLinkServices	Called from Setup to remove all instances of this link service.

The sample linkcfg.cpp DLL is written in C++ using the Microsoft Foundation Classes (MFC) and uses a single property sheet with two property pages as follows:

- The card configuraton property page implementation is in the cardcfg.cpp and cardcfg.h files. This property page is concerned with configuring various hardware properties (interrupt, DMA channel, and I/O address, for example) of the link service hardware.
- The connection mode property sheet implementation is in the mode.cpp and mode.h files. This property page is concerned with configuring mode information (link service name, link service title, SDLC line type, for example) for the link service.

The two property pages are linked to the link service property sheet in linkcfg.cpp within the **ConfigureLS** routine. This function is called by the exported **ConfigureLinkService** and **ConfigureLinkServiceEx** routines in linkcfg.cpp. An actual link service configuration DLL developed from these sources may require more property pages depending on the information needed to configure the actual link service DLL.

The registry.h include file used by linkcfg.cpp contains a global definition of the registry entries required for the sample generic SDLC link service. The values in this structure will be modified to contain the actual information specified by the user. This structure is added to the registry when a new link service is configured, and this structure is removed when a link service is deleted. The registry values that a developer must modify include the Link Registry Base entry (LINKSERVICE is used in the sample include file), the name of the device driver root (GenSdlc is used in the sample include file and source code), and various software and service registry settings appropriate for the target link service.

Several of the exported link service DLL functions use a configuration buffer, the CONFIG_BUFFER structure defined in linkcfg.h. The format of any CONFIG_BUFFER used by developers must match the structure format of this sample file for the first three parameters. Other parameters may differ for a developer's version of the CONFIG_BUFFER structure based on the target link service.

The sample link service configuration DLL calls a set of general utility functions that are not specific to any target link service. These utility functions are included in a lnktools library (lnktool.cpp) that is linked in as an OBJ file. This lnktools library includes the following utility functions that are useful in developing link service configuration DLLs:

Utility function	Purpose
AddPerfmonCounters	Add perfmon counters for this link service.
bCreateService	Create a service on a computer.
bDeleteService	Delete a service on a computer.
bStopService	Stop a service running on a computer.
CheckForExistingLinkService	Check to see if a link service of this type exists with this title.
ConvertHexStringToDWORD	Convert a hexadecimal string to a DWORD value.
ExtractNextParameter	Get the next parameter from a buffer.
fAddRegistryEntry	Add a new registry value to the registry.
fCanWeAdministerRemoteBox	Determine if the user has administrative privileges on the remote computer.
fConnectRegistry	Connect to a remote computer's registry and return a handle to the remote registry.
fDisconnectRegistry	Disconnect from a remote computer's registry.
fFindAndReplaceString	Find and replace a substring within a string.
fFindString	Determine if a string exists within a string buffer.
fFindStringInMultiSZ	Find a string in a REG_MULTI_SZ string list and return entire string.

fQueryRegistryValue	Query a value from the registry.
fRegistryKeyExists	Test whether a registry key exists.
fRemoveRegistryEntry	Remove a registry key.
fRemoveRegistryValue	Remove a registry value.
fStringCompare	Determine if two strings compare.
LoadStringResource	Load a string from the string resource.
ParseNextField	Return the next field from a string.
RemovePerfmonCounters	Remove perfmon counters for this link service.
ReturnString	Return a pointer to a string resource string.

The sample source code for a generic SDLC link service configuration DLL (linkcfg) includes several functions that may be useful as sample code when developing link service configuration DLLs for other hardware. The following functions are included in the linkcfg.cpp source code that may be of use as examples:

Utility function	Purpose
bDetectNetworkCard	Detect the remote network card and return the card settings buffer for the sample generic SDLC link service.
bLastGenericDFTLinkService	Check for the last generic SDLC link service for the sample generic SDLC link service. This routine is used to determine if the GENSDLC Device Driver (if one existed) can be removed.
ConfigureLS	The common link service configuration function used by the sample generic SDLC link service.
fAddAllRegistryValues	Add all registry values for the sample generic SDLC link service.
fAddClassAndBindformRegistries	Add the "class" and "bindform" registry entries for the sample generic SDLC link service. The bindform and class registry entries can only exist for the first link service of this type.
fEnumerateEventLogSources	Enumerate the Event Log sources registry value for the sample generic SDLC link service.
fRemoveAllRegistryValues	Remove all registry values for the sample generic SDLC link service.
fReplaceAllRegistryValues	Replace all user-provided information in the registry data for the sample generic SDLC link service.
fReplaceRegistryData	Replace global registry data for the sample generic SDLC link service.
fReplaceRegistryKeyName	Replace global registry structure strings for the registry key name for the sample generic SDLC link service.
fSetupGlobalValues	Create or update all user-provided information in the registry data structure for the sample generic SDLC link service.
InitializeGlobalStructure	Initialize link service data contained in the global data structure for the sample generic SDLC link service.

Contents of Integrated Link Service Sample Kit on SNA Server

The \SDK\SAMPLES\LINKSERV\GENERIC directory on the Microsoft® SNA Server CD-ROM contains sample source files for a generic SDLC integrated link configuration DLL. These sample files include the following:

- LINKCFG, the source code in C++ using MFC for a configuration property sheet for the sample generic SDLC. This link service configuration DLL must export specific functions.
- LNKTOOLS, the source code for various utility functions commonly used by an integrated link service DLL.
- DETECT, the source code for detection code for a sample generic SDLC link service.

This sample source code is included for your reference. The communication between workstation management station and SNA Server is performed by RPCSVC.EXE, the SNA RPC service.

Windows INF-Based Setup

This section describes link service setup using the older Microsoft® Windows NT® Setup Engine, which was designed to maximize versatility and minimize the work required to produce robust, full-featured setup programs. The engine contains most of the functions that a setup program might need and provides a scripting language similar to Basic in which entire setup programs can be written. For more information about this language, see the reference manual *Microsoft Windows 3.1 GUI Setup Toolkit*, Chapter 3, "The .INF File."

The Microsoft® SNA Server version 2.11 main setup program used such a script (as did the setup program for older versions of Microsoft® Windows NT®). As part of setting up SNA Server 2.11, the user chooses and installs support for various kinds of link services. Each type of link service comes with its own script, which is called from the main SNA Server setup program as a subroutine to perform all the tasks required for setting up that particular link.

Installing any link service requires that some common steps be performed; beyond these, the installation process can vary widely among different link types. For example, while the IBM DFT link service setup script needs to install a device driver for the DFT card, the 802.2 link service relies on the existing Token Ring or Ethernet adapter and does not need to install a device driver—but the 802.2 link service setup script does have to install the DLC transport if it is not already installed. Having some degree of commonality allows all link service setup scripts to share the same basic format and keeps the interface to the main SNA Server setup script relatively simple.

Since each SNA link service is actually part of the machine's "network ensemble," the SNA Server setup script and the link service setup scripts are designed to work with the Windows NT Network Control Panel Applet (NCPA). Adapter cards, device drivers, and link services for SNA communications that have been installed and configured by the SNA Server setup program can be configured from the NCPA as well.

You are not required to write your entire link service setup program as an .INF script. The actual script might merely launch a custom executable program to perform the installation; in this case the script provides the interface to the main SNA Server setup script. In addition, you can create custom DLLs that contain code and resources for doing certain setup jobs. The Windows NT Setup Engine already has such a DLL (SETUPDLL.DLL in the SYSTEM32 subdirectory of the root directory for Windows NT), which contains many useful setup functions.

The available documentation for the Setup Engine, the scripting language, and the support DLL is sparse at best. However, much can be learned by examining the existing .INF files both for SNA Server Setup (in the \SYSTEM\HWSETUP and root directories of your SNA Server tree) and for Windows NT itself (in the SYSTEM32 subdirectory of the root directory for Windows NT). Note that the main entry point for an .INF script is the [Shell Commands] section. You can list the contents of the support DLL with the command:

LINK32 -DUMP -EXPORTS SETUPDLL.DLL

and search the scripts for calls to LoadLibrary and LibraryProcedure.

This section contains:

Setup File Structure

This section discusses the link service setup script and the structure and contents of the files it uses.

This section contains:

Link Service Setup Interface

Inside the link service setup scripts, the components that need to be installed are listed as options. Two types of options used are main options and additional options. The main option usually represents the link service itself, and the additional options represent the subcomponents that are needed (such as device drivers or other system services). For example, in the IBM DFT install script, the main option is IBMDFTLS (IBM DFT link service) and the only additional option is IBMDFTDD (IBM DFT device driver). These names are arbitrary, but should be chosen to reflect what is being installed. Each option has a number of other attributes, such as a title (in the example, IBM DFT Link Service Support), and a version, (for example, 2.3).

At link service install time, main SNA Server Setup enumerates all the .INF files (link service setup scripts) in a certain directory and calls each one in turn. Each script provides its main options and attributes. The titles from each main option are presented to the user in a list box. When the user selects a title, main Setup calls that setup script to install its main and additional options.

Each component can be configured to permit installation of one or many instances of itself, depending on the needs of the link service. This allows a device driver to be installed once and have only one link service running on it, as is the case for IBM DFT support, or to have several instances of a link service on one device driver, as is done for IBM SDLC support. You can also have one link service using several device drivers.

An SNA Server link service setup script usually requires three files to work properly: the script itself (.INF), the Setup Resource Library (.SRL), and the online Help (.HLP). The Setup Resource Library is a special DLL that contains the dialog boxes, icons, and other resources needed by a particular script. The online Help file is a standard Windows Help file, meant to be used with the Windows Help Engine. References to these files can be removed from the scripts if they are not needed; however, it is recommended that you place stubs for these files rather than removing the references.

Disk Layout

The link service installation system was written with the assumption that IHV link services would be installed from a disk during or after the initial SNA Server setup process. For each link service on a particular disk, main Setup looks in the root directory for its .INF, .SRL, and .HLP files, and expects to find a subdirectory of the same name as the .INF file (without the .INF extension) where all other files for that link service are kept. For example, a disk containing the link services FastLink and SlowLink would have in its root directory two .INF files (FASTLINK.INF and SLOWLINK.INF), two .SRL files (FASTDLG.SRL and SLOWDLG.SRL), two .HLP files (FASTLINK.HLP and SLOWLINK.HLP), and the directories FASTLINK and SLOWLINK. Other files for these link services (device driver binaries, libraries, configuration files, and so on) reside in their respective directories.

The Setup Resource Library

The Setup Resource Library (SRL) file has the .SRL extension and contains resources for the dialog box templates that Setup will use. This file is built in the same manner as a DLL but is renamed to minimize confusion. Setup loads the .SRL file once and keeps a handle to it; then when it needs to display a dialog box, Setup uses the stored handle and the dialog box template name to fetch the dialog box from the library.

The SRL can be built by using the template and makefile found in the \SETUP\DLGDLL directory on the Microsoft® SNA Server SDK CD-ROM. Following is a list and brief description of the files in this directory:

File name	Description
DLLINIT.C	Contains startup and initialization code for the SRL to be used by Setup as a resource. This file is common to every SRL and should not be changed.
MAKEFILE	This file should be used with NMAKE to build the SRL. It should be edited to ensure that the path and file names are correctly assigned.
TEMPLATE.DLG	Contains the actual definitions of what the dialog boxes look like, their size and position, their replaceable text, their buttons and other controls, where their icons and/or bitmaps are located, and so on.
TEMPLATE.H	Contains many of the constant declarations that Setup uses to recognize controls such as option buttons and check boxes. These constants can be added to but should not be modified.
TEMPLATE.RC	Lists all the resources to be compiled into the library—dialog boxes, icons, bitmaps, strings, and so on. It is the input file for the Resource Compiler (RC.EXE) and includes TEMPLATE.DLG.
TEMPLATE.RES	This is the binary output of the Resource Compiler and is converted to an object file (.OBJ) and linked with DLLINIT.OBJ to produce the SRL. This file is also the input for the Dialog Editor, which writes out new .RES, .DLG, and .H files if you change a dialog box.

Replaceable Text in Setup Resource Libraries

In the dialog box definition file (.DLG), most of the text strings in the dialog box start with an at sign (@). This is the mechanism by which a single dialog box can be used for multiple purposes inside a setup script. If the text is @Text1, for example, the script can set a symbol called Text1 to some arbitrary text before displaying the dialog box, and when the dialog box appears, the new text will occupy the position where @Text1 appears in the dialog box definition. This also applies to list box titles, button labels, and so on.

One good use for this feature is error handling — a single dialog box with caption @ErrorString, text @ErrorDesc, and buttons @Button1 and @Button2 could be used to display information about many different errors. For example, set ErrorString=Fatal Setup Error and ErrorDesc=Some Important File Is Corrupted. The buttons at the bottom of the dialog box could be changed by setting Button1=Ignore Error and Button2=Try Again. See the existing .INF scripts for examples.

The Online Help File

The context-sensitive online Help (.HLP) file must be a standard Windows Help file. It is written in Rich Text Format and compiled into WINHELP format. The first page in this file should be a contents page, and it should contain a page for each context-sensitive Help message required. When the user presses the **Help** button in a dialog box, the Setup Engine calls WINHELP with the script's Help file and the *HelpContext* identifier. See the existing .INF scripts for examples.

Creating an INF-Based Setup Script

This topic describes the recommended sequence of steps to follow when creating a setup script for an IHV link service. It will be helpful to refer to existing .INF files while reading this discussion.

Create the Initialization Section

The first section in the setup script is the initialization section. It contains most of the constant text strings and other values that describe the link service and its components. It is important that a clear layout of the SNA options and their additional components is created in this section.

Define the Parameters

Define the parameters for each component and how they fit into the registry. It is generally a good idea to name the variables for these parameters after their names in the registry, as this simplifies the code and greatly improves its readability.

Complete the Basic Code

Fill in the code sections in the main body of the setup script. Write only the code to initialize the components and write their parameters to the registry. Since modification of variables involves reading their values from the registry, it can be ignored until this step is complete and the information in the registry is correct.

Design Dialog Boxes

Copy the Setup Resource Library template files into another directory, then change the file names from TEMPLATE.* to *WHATEVER*.* (use whatever name you want). Using the Dialog Editor, add your own dialog boxes that will be used to modify the parameters. Make sure the dialog box layouts are clear and intuitive. Consider which dialog boxes will require online Help. Also, remember that controls should be accessible by tabbing.

When your dialog boxes are complete, edit the makefile and .RC file to make sure all paths and file names are correct. Type NMAKE to build the new SRL.

Fill in the Code

Fill in the code sections to modify the variables. This includes writing the dialog box-handling code as subroutines in the script and writing the sections that call those subroutines from the main body of the script. This step is probably the most complex. The existing link service setup scripts provide examples.

INF-Based Setup Template Description

This section describes options, variables, and entry points used in the setup scripts using INF files.

This section contains:

Initialization Section

This section describes the options that control the general behavior of the script.

Source Media Descriptions

The Source Media Descriptions section contains the labels for the disks (media) used by the setup script for its installations. For more details on this section, you can consult the chapter on .INF files in the *Microsoft Windows 3.1 GUI Setup Toolkit* reference guide. Note that additional parameters on the description lines (for example, TAGFILE) must be in uppercase, and the spacing must be exactly as used in the existing .INF files.

Option Type

The option type is used by the Network Control Panel Applet (NCPA) to determine what kind of option this setup script can install. The NCPA expects each script to support only one type of option. The setup scripts described here use the Network Service component type (abbreviated NetService) because they need to install multiple component types and NetService is the most general type.

Languages Supported

This section should contain a list of human languages supported by the setup script. Each language is represented by a three letter abbreviation, the most common of which is ENG for the English language. Throughout the setup script, each section that has a language dependency should be rewritten once for each language. Examples of this usage abound in the existing .INF files.

> **Note** The SNA Server setup scripts currently support only the English language. While it is possible for an IHV script to support its own set of languages, before calling any of the utility functions you must set the language specifier to English.

Option List

The option list is the most important subsection of the initialization section. It lists all of the options that a particular setup script can handle. When the NCPA queries this script, it expects to be able to install each of these options.

> **Note** Although the SNA Server setup scripts allow link services to be inspected and configured using the NCPA, they do not support installation by the NCPA at this time.

Options Text List

This is a list of names or titles for each of the options specified in the option list. When the NCPA or SNA Server Setup calls this script, these are the names that are presented to the user. Since names are language-dependent, there should be a list of names for each language supported.

SNA Option List

The SNA option list is a subset of the option list. It lists all the options that SNA Server Setup will offer the user.

SNA Additional Options

When SNA Server Setup needs to install one of the options available to the user, it first tries to install these options. SNA Server Setup reads this list of additional components and installs each of them in order before it installs the main component. When SNA Server is removed, this list of components is used in the removal of a particular support system. The SNA Server Setup loops through its list of installed options, calls this script for the list of additional components, and attempts to remove each of these components as well as the main option.

Installation Steps

Two steps are involved in copying files during the installation of a component:

1. The script adds files to a list of files to be copied.
2. The script calls a function to copy the files in the list.

These options specify which of the two steps will be executed for each component's installation. These options apply only to installation by the SNA Server Setup. The NCPA always executes both steps of the file copying.

These options allow you to perform all of the file copying for all the components at the end of the installation. This can be accomplished by making the main component the only component that actually copies the files in the list. All other components add their files to the copy list until the main component copies all of the files at once.

Dialog Box Constants

This section defines many of the major dialog box constants. It contains the variables used for displaying the file copy progress gauge, as well as those for displaying the context-sensitive Help. This section also defines variables to represent option buttons, check boxes, and dialog box entities.

Progress Copy Variables

These variables define the look and behavior of the progress gauge displayed during file copies.

Variable name	Description
ProCancel	The text displayed in the **Cancel** button of the gauge window.
ProCancelCap	The caption of the cancellation dialog box.
ProCancelMsg	The message that appears in the cancellation window that appears when the user presses the **Cancel** button.
ProCaption	The caption of the gauge window.
ProText1	The label that precedes the source file name.
ProText2	The label that precedes the destination file name.

Help Context Identifiers

The *HelpContext* identifiers are used to identify which dialog box has had its **Help** button pressed. When the user presses the **Help** button, Setup calls WINHELP with the identifier assigned to the dialog box being displayed. These variables can be modified by individual IHV setup scripts.

File Constants

This section describes variables that provide file names, constants, and product information

File Names and Initialization Constants

These variables define file names and other constants.

Variable name	Description
HlpMax	This number should be set to one more than the maximum value of the context-sensitive Help identifiers.
HlpMin	This number should be set to one less than the minimum value of the context-sensitive Help identifiers.
ShellCode	This variable holds the error return code from all shell executions and is set by the Setup Engine itself. A value of zero represents success; all other values represent errors.
SubroutineInf	This .INF file contains some of the utility functions used by both SNA Server Setup and Windows NT Setup.
ThisFile	This variable should hold the name of this .INF file without the extension. Defining it here is the only way to access this information in the setup script.
ThisHlp	The full name of the WINHELP file for this setup script.
ThisInf	The full name of the setup script.
ThisSrl	The full name of the Setup Resource Library for this script.
UtilityInf	This .INF file contains all of the SNA Server Setup utility functions.

Product Information

This section contains all the information about a single component. There should be a section of this form for each of the options supported by this script.

Variable name	Description
FullInfName	The full path to this setup script (.INF file) after installation.
NetRulesClass	The class to which this product belongs for bindings determination by the NCPA.
ProductDepends	This variable holds the dependencies of this product. This value is only used in the creation of the product service key.
ProductDll	The dynamic-link library that should be linked to by the control APIs for this component. This variable is only necessary for link services and should be given a null value for all other cases.
ProductExclusive	This variable dictates the behavior of multiple attempts at installing this component. It can hold one of three values: • FALSE: This product is not exclusive and can be installed as many times as the user chooses. • TRUE: This product is exclusive and can only be installed once. • NOTIFY: This product is exclusive, and the user should be informed of this fact. SNA Server Setup displays a message to inform the user.
ProductExtraParams	This variable holds the entries for the **ExtraParameters** subkey of the product service key. It is recommended that any modifications to the value of this parameter be done in the body of the setup script with the rest of the assignments. This line serves as the initialization for this variable.
ProductFullName	The actual value used to represent this product in the registry. It is a combination of the product name and version, separated by an underscore.
ProductImagePath	The file name of the executable image for this product. This can be an executable file or a system file. For a link service, this variable must always contain the value SNALINK.EXE.

Product Name	The name of this product. This name is used in conjunction with the version to provide the full name of the registry entry for this product.
Product Params	This variable holds the entries for the **Parameters** subkey of the product service key. It is recommended that any modifications to the value of this parameter be done in the body of the setup script with the rest of the assignments. This line should not be deleted since it serves to initialize this variable.
Product RegBase	The specific location for this product's registry entry under the **SOFTWARE** registry key. This variable should not be changed since all the SNA Server components should reside under **SOFTWARE\Microsoft** in the registry.
Product Service Prefix	A prefix that is used to algorithmically decide the service name for this component. Service names for SNA Server are limited to no more than eight characters. Since SNA Server Setup attaches a single-character suffix based on the index, this prefix should be no longer than seven characters.
Product Type	An internal representation of the type of component this option belongs to. Currently, this variable can take on the values of Link and Driver. **Note** This value is currently unused in the SNA Server scripts; however, it should still be set since its usage will be implemented in the future.
Product Version	The version of this product (component). This number is used in creating the registry key for this product under the **SOFTWARE** key of the registry.
Service Type	This variable is used to declare the type of this component for usage with the NCPA bindings.
SNAServiceType	A more specific description of the type of component this option represents. It should be set to one of the values specified in the SNAServiceTypes section of the *UtiltyInf* file.

SNAServiceType Values

The decimal value for the **SNAServiceType** registry entry for each of the link service types is as follows:

Link service type	SNAServiceType value
802.2	11
Channel	32
DFT	10
SDLC	3
Twinax	31
X.25	4

A link service for Twinax works like a link service for DFT, and therefore does not contain any link-specific data.

The 802.2 link service can be for either Ethernet or Token Ring.

General Constants

This section defines some general constants that are used in the setup scripts. You can add variables to this section, but you cannot modify or delete any of the existing ones.

Flow control variables	Description
from	The label that the setup script last passed.
to	The label that the setup script is headed for.

State variables	Description
TRUE	Boolean true.
FALSE	Boolean false.
NOTIFY	Used in the *ProductExclusive* variable defined in File Constants.

Registry initialization	Description
NoTitle	Used in the creation of registry keys. This variable specifies that the key to be created should not be assigned a title. None of the keys created by SNA Server setup scripts require a title. For more information, see documentation on the Windows NT registry.
KeyNull	The null key handle. It is most often used for comparisons and initializations of other handles.
KeyProduct	Handle to the product key in the **SOFTWARE** registry tree. It is assigned here for initialization purposes.
KeyParameters	Handle to the **Parameters** subkey of the product service key. It is assigned here for initialization purposes.

Additional initialization parameters	Description
ExitState	This variable holds the state of the script after it exits. Note This feature is currently unused.
OldVersionExisted	A Boolean value that describes whether a previous version of this product exists.

Language-Dependent Dialog Box Constants

This section contains dialog box variables and constants that are language-dependent. Any dialog box information that is language-dependent should be placed here. This section contains a list of the English texts for the basic dialog box buttons. This list should not be modified, as consistency should be preserved across SNA Server setup scripts.

Language-Dependent File Constants

This section contains language-dependent variables and constants that relate to file information. The *SetupTitle* variable should be defined because it is used in some captions. Also, this section should include a subsection for each option and each language containing two variables each: *ProductTitle* and *ProductDesc*. These variables contain the default values for the title and description of this product. The user can modify the title of the product, but not the description. The three variables mentioned here can be modified by individual IHV setup scripts.

Date Section

This section is used to determine the date of installation. This information is currently unused by SNA Server Setup but can be useful for some purposes.

Input Dialog Box Information

This section should contain all information about the input dialog boxes. Each dialog box to be displayed should have a subsection that contains its variables. There should be a subsection of this kind for each language supported, since dialog box text is generally language-dependent. For examples, refer to any of the existing setup scripts.

Input Dialog Box Scripts

This section should contain all the code for handling the input dialog boxes. Each script can return as many parameters as needed, as long as the first parameter is the overall status and is one of the standard status return values.

Identification Functions

The functions in this section are used by SNA Server Setup and by the NCPA to gather information about this script and the options it manages.

Identify

This function is called only by the NCPA and returns the most basic information about this script. It tells the NCPA what type of option this script installs as well as the media it requires. The NCPA expects the option type returned by this function to represent all of the options in this script. The setup scripts here will use the most generic type, NetService.

Return Options

This function returns a list of the options supported by this script and their appropriate text strings. The text is assumed to be language-dependent and will be drawn from one of the OptionsText sections.

Return SNA Options

This is SNA Server Setup's parallel function to the Return Options function. It returns only those options that SNA Server should display to the user. These are specified in the initialization section.

Return SNA Additionals

This function takes as a parameter the option it should query and returns the list of additional components for that option. It also returns a list of the text strings for those additional components.

SNA Invocation Section

When SNA Server Setup needs to call this script, it does so through this entry point. By setting the variables *NTN_InstallMode* and *NTN_InfOption*, SNA Server Setup can control the behavior of this script. This entry point also expects the *STF_LANGUAGE* variable to be set to a language the script can support.

NCPA Invocation Section

When the NCPA needs to call this script, it uses this entry point to do so. As with the entry point in the preceding topic, the NCPA sets the appropriate variables before calling this entry point. This entry point also sets up some other variables that the SNA Server Setup would normally create. Among these variables are *SNARootDir* and *SNAVersion*.

Common Code Section

This section contains most of the actual code that the script runs. It expects to receive the name of an option and an index as parameters. It also checks to see that the language specified in *STF_LANGUAGE* is supported. It then checks the *NTN_InstallMode* variable to determine what it should do. Before proceeding to the section of code that will complete the required task, the script sets up default values for all the variables of the particular option being installed.

Main Entry Section

This section checks the value of the *NTN_InstallMode* variable and determines the label of the code that will complete that operation. It also assigns defaults values for the variables.

Set Defaults (set_defaults)

This section sets the default values for the options; among them are the service name, title, and description. These three must be set in this section; however, more variables can be added as needed.

Install Component (install_component)

This section checks to see if there are any instances of this component already installed. If there are none, it sets the install index to one and continues. Otherwise, if one instance exists, it checks for the value of the *ProductExclusive* variable to determine what it should do. If it needs to install another instance of the component, it executes the utility file and finds the next available index to use.

When it is ready, the script determines the service name for this component using an algorithmic definition (install_nextstep). If the *ProductServiceQuery* variable is set to TRUE, the user is presented with this value and is allowed to change it. When all the variables are ready, it proceeds to the parameter modification section.

Configure Component (config_component)

This section determines whether or not the component in question is installed. If the component is not installed or if the index is invalid, it fails with an error. Otherwise, it reads the parameters for this component from the registry (**read_params**). This code reads the parameters from the **Parameters** and **ExtraParameters** keys in the registry and creates a list of their values. It then loops through the list and assigns the values to the variables using a switch/case construct (**assign_value_*option***). When that is done, there is an extra section (**assign_extra_*option***) where other parameters that do not reside in the standard location can be read or any other code that assigns values to variables can be placed. When the variables are all assigned, the code proceeds to the parameter modification section.

Modify Parameters (modify_params)

This section contains the code that modifies the variables or calls out other code that modifies them. As soon as this section is entered, control is routed to subsections, one for each option (**modify_params_*option***). These subsections should contain all of the actual modification code. When this section is done, the variables are ready to write to the registry and the code proceeds to the adjustment section.

Adjust Parameters (adjust_params)

This section takes control from the parameter modification section. If the script is supposed to configure an existing component, the control is routed directly to the code that writes the parameters to the registry. Otherwise, control falls to code that creates the appropriate service entries in the registry before it writes out the variables.

Create Registry Entries (create_regvals)

This section creates the service and registry entries for this component. It also writes out some of the values that the product entry under the **SOFTWARE\Microsoft** key needs to contain. This section is also currently in charge of the NCPA bindings information. If the product files have not been copied, this code shells out to the installation section that handles file copying (**InstallRemove**). When all the keys are created, control falls to the section that writes out the variable values to the registry.

Write Out Variables (write_params)

This section first ensures that the appropriate handles are open and then routes control to a subsection that prepares the appropriate variables for registry output (**write_params_*option***). Each of these sections must prepare the *ProductParams* and *ProductExtraParams* variables to be written to the registry. Each of these parameters is a list of registry creation lists. Each registry creation list must contain the name of the entry to be created, the *NoTitle* variable, the type of entry to create, and the value of the entry. See the existing link service .INF scripts.

The *ProductParams* variable must contain the name and option entries. Both variables can contain any additional entries that are needed. Also, any other parameter-writing code should be written in these sections. For an example, see the setup script for the NDIS 802.2 link support (SNADLC.INF).

After the subsections are finished, control is passed to the code that actually adds the list of entries to the registry. It also prepares the data return structure and finishes by using the successful escape hatch.

Get Bindings From Component (getbind_component)

This section is responsible for communicating the option bindings back to SNA Server Setup. It queries the NCPA bindings, sets the information in the data return structure, and returns it to SNA Server Setup. SNA Server Setup then uses this information to determine whether this particular instance can be removed or not. An instance can be removed if it is not needed (bound to) by any other component.

Remove Component (remove_component)

This code is in charge of removing one or all of the instances for a particular product. In the case of a complete removal, it also removes the files and software entries for this product. In the case of a single instance removal, the code passes control to the **remove_one_piece** subsection, which handles a single removal. Otherwise, code control goes to the **remove_all_pieces** subsection.

For a full removal, the setup script loops through all of the instances and calls the remove_one_piece subsection for each of those instances. When the loop is complete, the code control is passed to a common point for both removal types (remove_product). The subsection remove_one_piece removes the software entry for the instance and deletes the service from the Service Control Architecture. If it is dealing with a complete removal, it returns control to the loop, otherwise, it falls to the remove_product subsection. This subsection determines whether files should be removed (full removal) and calls the appropriate installation section (InstallRemove). After all is done, it uses the successful escape hatch to return control to SNA Server Setup.

Escape Hatches

These are all the available hatches that can be used throughout the setup script.

Successful (successful)
Sets the status to *STATUS_SUCCESSFUL* and exits through **end**.

Warning Message (warning_msg)
This hatch displays a warning dialog box with the error message stored in the variable *Error*. This variable should be defined before calling this escape hatch. This warning dialog box has two dialog buttons: **OK** and **Cancel**. If the user chooses to continue, control is passed to the *to* label, otherwise, control is returned to the *from* label. If the warning box fails, the script exits through **end**.

Nonfatal Message (nonfatal_msg)
This hatch displays a nonfatal warning dialog box with the error message stored in the variable *Error*. This variable should be defined before calling this escape hatch. This dialog box has only one button, **OK**. After the user presses this button, control is returned to the *from* label. If the nonfatal warning dialog box fails, the script exits through **end**.

Fatal Registry Message (fatal_registry)
This is probably the most-used escape hatch, next to the **successful** hatch. It sets up the *Error* variable with a template that includes three other variables: *ErrMesg*, *ErrProc*, and *ErrFunc*. The first should be a short message describing the error. The

second should be the section or subsection that called the registry function. The third should be the name of the registry function that was called. All three variables should be set before calling this escape hatch. After the error message is prepared, the **fatal_msg** escape hatch gets control.

Fatal Message (fatal_msg)

This hatch displays a dialog box with the fatal error message. When the user clicks the button, this hatch exits through the **set_status_failed** escape hatch.

Shell Code Error (ShellCodeError)

This hatch is only used when a shell execution fails. If the script receives a *ShellCode* that is not zero, it calls this hatch. This hatch displays an error message and exits through the **set_status_failed** hatch.

Failed Exit (set_status_failed)

This hatch sets the status to *STATUS_FAILED* and exits through **end**.

End

This hatch ensures that the handle to the SRL for this file is closed and exits this script. It should not be called directly without ensuring that the *Status* variable contains the correct return value for the operation.

Installation Control Section

This section contains the code that copies and removes files for the scripts.

Install Section

In the case of an installation, this section reads in the file list to be copied and copies it, depending on the values of the appropriate variables (see the initialization section.) The script copies files into the SYSTEM and SYSTEM\HWSETUP subdirectories. You can add files to the file list by calling the same function that the script calls.

In the case of a removal, the script reads in the same list of files that it used for the copy and creates a list of file names in the global variable *_FileRemoveList*. Each of the file names in this list is specified by its full path. SNA Server Setup then tries to remove all the files on this list. This layer of abstraction is necessary because the SRL file cannot be deleted while a handle to it remains open.

File List Section

This section specifies the file list for each of the options supported by this setup script. Options may or may not have a section for files to be copied into the SYSTEM and SYSTEM\HWSETUP subdirectories. If a section listing files exists, those files will be added to the copy list as needed. The three main files for each script (.INF, .SRL, and .HLP) should be placed under the SYSTEM\HWSETUP subdirectory and invoked by all the SNA options in this script.

In the case where more than one SNA option exists, the script uses version numbers to determine whether or not the files should be recopied. If the version numbers are the same, the files are not recopied.

Global Variables

The following table describes global variables for the setup scripts using INF files.

Variable name	Description
AddCopy	Boolean indicating whether files should be added to the copy list.
DoCopy	Boolean indicating whether files should be copied at all.
DriverDir	Path wherse link service .INF files are found.
IHVDLGHANDLE	Handle to SRL containing dialog boxes.
LF	Constant for printing newlines in dialog box text.
LIBHANDLE	Handle to the Setup Support Library, SETUPDLL.DLL.
NTN_InfOption	Which option is currently being installed/removed.
NTN_InstallMode	Controls installation/configuration/removal behavior.
NTN_SoftwareBase	Handle to **SOFTWARE\Microsoft** registry tree.
REG_H_LOCAL	Handle to top of **HKEY_LOCAL_MACHINE** registry tree.
REG_KEY_READ	Constant for read-only registry access.
REG_KEY_READWRITE	Constant for read/write registry access.
REG_VT_SZ, etc.	Constants representing data types for registry entries.
SNARootDir	The root directory of the machine's SNA Server tree.
SNAVersion	Current version of the SNA Server.
STF_LANGUAGE	Which human language is currently in use.
STF_WINDOWSSYSPATH	Path to Windows NT system subdirectory.

Utility Function Overview

The following table summarizes some of the useful entry points in the .INF file that contains utility functions. The file name is in the variable *UtilityInf*; usually set to SNAUTILS.INF.

Function name	Description
CreateSNARegEntry	Creates the necessary entries for an instance in the **SOFTWARE\Microsoft** registry tree.
CreateSNAService	Creates the necessary entries for an instance in the **Services** registry tree.
DeleteSNAService	Deletes a particular service using the Service Control Manager.
EnterServiceName	Presents the user with an algorithmically determined service name for a component and allows the user to change it before returning the final value.
FindNextAvailableIndex	Determines the index a new instance should receive.
FindSNAProductServices	Enumerates all instances of a product.
FindSNARegEntry	Attempts to open all of the necessary registry keys and return open handles to them.
FindSNAService	Provides an easy way to access the keys for a particular service.
GrepUniqueServiceInfo	Determines the information about a particular instance when only one of the four elements is available.
SetupMessage	Displays a dialog box with user-defined text plus **OK** and **Cancel** buttons.

Diagnostics

This section describes the following diagnostics mechanisms available to Microsoft® Host Integration Server 2000 and SNA Server applications.

This section contains:

- Error and Audit Log Messages
- Internal Tracing
- DLC Tracing
- ConnectionTracing

In addition, there is a section describing how the IHV can provide a link level tracing facility similar to the standard Host Integration Server 2000 or SNA Server SNALinks.

Error and Audit Log Messages

This section discusses ways that an application can write to the Microsoft® Windows 2000 or Windows NT® Application event log, and describes macros for logging and tracing information.

Options for Logging

The Microsoft® Host Integration Server 2000 and SNA Server header file TRACE.H provides macros that can be used to write to the Windows 2000 or Windows NT Application event log files.

On Host Integration Server 2000 and SNA Server 4.0, these macros use the message file that is linked into SNAEVENT.DLL, the central logging DLL that contains every log message in the Host Integration Server 2000 or SNA Server system. The IHV has two options when logging an error message:

- If the standard Host Integration Server 2000 or SNA Server product contains a log message that describes the error condition, this can be used by the IHV.
- If extra messages are required, the generic log messages COM0393 and COM0394 can be used. Each takes two parameters, both of which are text strings: the first is an identifier for the SNALink that logged the message, and the second can contain any data or parameters to be logged. The difference between these messages is the level at which they are logged; COM0393 is a level 10 information message, while COM0394 is a warning message that should be used to report error conditions. See Host Integration Server and SNA Server DMOD Logging Macros for more information on message severity levels.

Host Integration Server and SNA Server DMOD Logging Macros

The logging macros provided with Host Integration Server 2000 and SNA Server are relatively easy to use because each log call is a single line of code. In the simplest case, only a message number is required; the text of the logged message is taken from the message file. Parameters can also be supplied as required.

For the text and meaning of each of the log messages included in the Host Integration Server 2000 message file, see the *Administrator's Reference*. Examples of messages logged by the 802.2 DLC SNALink supplied with Host Integration Server 2000 and SNA Server are shown in Examples.

Message Severities

The following severities are used in Host Integration Server 2000 and SNA Server:

Severity	Description
6	Detailed problem analysis data
8	General information messages
10	Significant system events
12	Warnings/recoverable errors
16	Fatal errors

All logs are placed in the Windows 2000 or Windows NT Application event log. The default level of audit logs can be specified when configuring Host Integration Server 2000 or SNA Server, so that all or some of the audit logs can be suppressed and lower-level messages can be filtered out when viewing log files using the Windows 2000 or Windows NT Event Viewer program.

Note that level 16 errors are always taken to be fatal errors; an application that logs a level 16 error will be terminated automatically. In particular, the call to the logging routine will not return; the application should perform all required cleanup processing, including any lower-level logs, before logging the fatal error.

Logging Macros

The following macros can be used to log messages at levels 6, 8, 10, 12, and 16:

COM_LOG6
COM_LOG8
COM_LOG10
COM_LOG12
COM_LOG16

Syntax

COM_LOG*a* (*b* **)** "" **))**
for a message with no parameters.
COM_LOG*a* (*b* **)** " %*c* " , *e* **))**
for a message with one parameter.
COM_LOG*a* (*b* **)** " %*c* | %*d* | . . ." , *e* , *f* , . . .**))**
for a message with more than one parameter.

Parameters

a
Severity: 6, 8, 10, 12, or 16.
b
Message number.
c, *d*, . . .
Format of the first, second, and so on up to nine variable parameters.
e, *f*, . . .
First, second, and so on up to nine variable parameters.

Remarks

Up to nine parameters can be supplied, according to the number of "*%n*" placeholders in the text of the message being logged. The first parameter replaces %1, the second replaces %2, and so on.

The formats *c* and *d* must be valid formats for the C function **sprintf**, because the logging macro uses this function to generate the complete text string to be logged.

Note that the unmatched parentheses on the macro call are deliberate; the expansion of the macro supplies the remaining parentheses so that the resulting code is correct.

Examples

This syntax of the logging macros is illustrated in the following examples. The two messages below are taken from the Host Integration Server 2000 and SNA Server message file; the first is a warning message with no parameters, and the second is a level 16 error message that includes one parameter.

```
COM0236W: DM or DISC received
COM0242E: Invalid adapter (%1) configured.
```

The following call:

```
COM_LOG12(236)""));
```

gives the following message in the error log:

```
DM or DISC received
```

The following call:

```
COM_LOG16(242)"%d", adptr));
```

where *adptr* = 17, gives the following message in the error log:

```
Invalid adapter (17) configured.
```

The complete message log entry contains the following information about the error or event and the service that logged it:

Date/time
User name
Computer name
Message number (event 10)
Source
Type (severity)
Message with optional parameters

The source is the name of the link service that generated the event log; for example, SNA SDLC link service.

Internal Tracing

Microsoft® Host Integration Server 2000 and SNA Server provide internal tracing macros that can be built into an application. These can be used to check the operation of the application during development. Compile-time options determine whether or not this tracing is included in object code; it should be compiled out when producing end-user products. Internal Tracing Macros and Controlling Internal Tracing describe the use of the tracing macros within application source code and the methods of controlling trace output using compile-time and run-time options.

> **Note** Tracing must be initialized by the IHV DLL before any **TRACEnn** macros are used. The TRACE.H header file is included with the Host Integration Server 2000 and SNA Server SDK. Use the **INITIALIZE_TRACING** macro, listed in the TRACE.H header file, to initialize tracing. The best place to issue the **INITIALIZE_TRACING** call is at the start of the SNALinkInitialize routine.

Internal Tracing Macros

The following topics describe macros that are used for internal tracing:

- COM_ENTRY
- TRACEn

COM_ENTRY

The **COM_ENTRY** macro is used at the start of any procedure in which internal tracing is required. It provides an identifier that is used in the trace file to identify all trace calls made from this procedure.

The format of the **COM_ENTRY** call is as follows:

COM_ENTRY("*str*"**)**;

where *str* is a string of up to five characters that uniquely identifies this procedure.

TRACEn

The **TRACE*n*** macro is used to specify data to be traced; this data can include variable parameters. The format of the **TRACE*n*** call is one of the following, depending on whether parameters are included:

TRACE*n*()"*string in **sprintf** format*"**))**
TRACEn()"string in **sprintf** format containing parameters," parameters**))**;

Note that the unmatched parentheses are deliberate; they are resolved by the expansion of the macro.

The value *n* identifies the severity level of the trace. It can take the values 2, 4, 6, 8, 10, 12, or 16, where levels 6 to 16 correspond to the audit and error log levels (see Error and Audit Log Messages), and 2 and 4 are used for very low-level detail tracing. This allows run-time filtering of trace information so that only information above a specified level is logged; see the following section for more information.

The following examples illustrate the use of the tracing macros:

```
COM_ENTRY("proc1");
TRACE4()"Start of error-handling routine"));
TRACE8()"Supplied parameters are %s, %d", parm1, parm2));
```

Controlling Internal Tracing

The compiler option **NOTRC** defines whether internal tracing is included in object code. Compiling with **/DNOTRC** does not include internal tracing (the **TRACE*n*** macros expand to a no-op); compiling without **/DNOTRC** includes internal tracing.

> **Note** The registry entries used to control tracing are inserted by the Setup program. The SNATRACE.EXE program can be used to enable or disable internal tracing dynamically at run time (assuming binaries have been compiled with internal tracing enabled).

When running an executable program that was compiled with internal tracing, tracing is enabled by generating the following entries in the Microsoft® Windows 2000 or Windows NT® registry:

InternalTraceLevel=*n*
InternalTraceFile1=*file1*
InternalTraceFile2=*file2*

FlipLength=Length_in_Bytes

The entries should be stored under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services
\<SNALink name>\Parameters

The value *n* is the severity level of the tracing required. All trace calls at this level or higher are included in the trace output; trace calls at lower levels are ignored. For example, setting level 10 includes all trace calls at levels 10, 12, and 16, but excludes tracing at levels 2, 4, 6, and 8. Use 0 to include all tracing at whatever level, or 20 to disable tracing entirely.

The parameters *file1* and *file2* are the names of files to which trace output is written. If two file names are specified, trace output is sent to the first file until it reaches **FlipLength** bytes and then is sent to the second file; when the second file also reaches **FlipLength** bytes, the first file is cleared and tracing continues to the first file. This process continues, changing to the other file every time the current file reaches **FlipLength** bytes, so that only the most recent **FlipLength** bytes of trace information is retained. If only one file name is specified, **FlipLength** is disregarded and tracing continues to this file regardless of file size.

DLC Tracing

Microsoft® Host Integration Server 2000 and SNA Server provide the facility for tracing message flows at the Data Link Control (DLC), both at the local node and at the application's Base. This allows you to track the messages being sent and received by the local node and the SNALink.

> **Note** The registry entries used to control tracing are inserted by the Setup program. The SNATRACE.EXE program can be used to enable or disable internal tracing dynamically at run time (assuming binaries have been compiled with internal tracing enabled).

Message tracing at the SNALink is controlled by entries in the Microsoft® Windows 2000 and Windows NT® registry under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\...\<SNALink name>\Parameters

Message tracing at the local node is controlled by placing similar entries under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\SNAServr\Parameters

The entries required in both cases are as follows:

MessageTraceFile1=*file1*
MessageTraceFile2=*file2*
DLCTraceState=on (*or* off)

Connection Tracing

All SNALinks supplied with Microsoft® Host Integration Server 2000 and SNA Server provide connection tracing at their lowest level. Examples of this information are:

- Parameter blocks passed to/received from the CCB2 interface by the 802.2 DLC SNALink.
- Frames sent/received on the line by the X.25 and SDLC SNALinks.
- Command frames sent to/received from the controller by the DFT SNALink.

It is recommended that IHVs also provide this type of tracing to aid troubleshooting when the SNALink is in operation at remote customer sites.

Because the lowest level of the SNALink can vary greatly, according to both the type of the link being serviced and the features provided by driver software being used, it is not feasible for Host Integration Server 2000 or SNA Server to provide a tracing interface that will format raw data passed to or from the adapter.

The Connection Tracing interface provided for IHV use requires the SNALink to provide a preformatted buffer of ASCII text that will be written to a trace file maintained by Host Integration Server 2000 or SNA Server. This allows the IHV SNALink to trace to the same trace file as other SNALinks but leaves Host Integration Server 2000 or SNA Server in control of access to the file.

COM_TRC_IHV

The **COM_TRC_IHV** macro specifies the buffer to be output to the trace file. It is invoked using

COM_TRC_IHV(&Buffer);

The buffer must be formatted as multiple lines of ASCII text (a maximum of 18 lines), each exactly 60 characters in length. Carriage returns and line feeds should not be used. The end of the buffer should be signaled by an ASCII NULL in the data. The buffer will be output to the file in the standard Host Integration Server 2000 format, as shown in the following figure.

```
| PP.TT IHV    ------------------------------------------- Time
| PP.TT IHV    Buffer [0..59]
| PP.TT IHV    Buffer [60.119]
| PP.TT IHV    .
| PP.TT IHV    .
| PP.TT IHV    ------------------------------------------- Time
| PP.TT IHV    Buffer [0..59]
| PP.TT IHV    Buffer [60.119]
| PP.TT IHV    .
| PP.TT IHV    .
| PP.TT IHV    ------------------------------------------- Time
| PP.TT IHV    Buffer [0..59]
| PP.TT IHV    Buffer [60.119]
```

Format of connection tracing. PP is the process identifier and TT is the thread identifier.

> **Note** The registry entries used to control tracing are inserted by the Setup program. The SNATRACE.EXE program can be used to enable or disable internal tracing dynamically at run time (assuming binaries have been compiled with internal tracing enabled).

IHV connection tracing is enabled by including the following entries in the Microsoft® Windows 2000 or Windows NT® registry:

ConnectionTraceState=on *(or* off*)*
ConnectionTraceFile1=*file1*
ConnectionTraceFile2=*file2*

These entries should be stored under:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\...
Services\<SNALink_Name>\Parameters

For further details of connection tracing, refer to the Host Integration Server 2000 or SNA Server documentation.

Compiling and Linking a SNALink

This section also provides information on compiling and linking a SNALink for use with the Microsoft® Host Integration Server 2000 and SNA Server. This section also lists and explains the header files and libraries need to build a SNALink.

Host Integration Server and SNA Server DLC Header Files

The following files are required to build a Microsoft® Host Integration Server 2000 or SNA Server SNALink:

SNA_DLC.H	Main header file containing the definitions of buffer and message formats.
SNA_CNST.H	Function prototypes for the Base/DMOD interface calls and constant definitions.
TRACE.H	Definitions of the logging and tracing macros (see Diagnostics for more information).
IHVLINK.LIB	Main library for the SNADIS interface.

Included Files

To compile a SNALink, the header files SNA_DLC.H, SNA_CNST.H and TRACE.H are required. In addition, one of the standard operating system header files may be required. To include the required files, the following lines should be used in your application:

```
#include <sna_dlc.h>
#include <sna_cnst.h>
#include <trace.h>
```

Note that the TRACE.H include file is required to enable SNA tracing and use of the Host Integration Server 2000 or SNA Server Trace viewer utility.

Required Exports

The IHV link support DLL must export the following entry points:

SNALinkInitialize
SNALinkWorkProc
SNALinkDispatchProc

These are called by the Base scheduler when the SNALink is invoked.

Compiler Options

When compiling the SNALink DLL, the following compiler options are required:

Option	Explanation
/c	Compile only, without linking. Linking is done as a separate phase to include the required Microsoft® Host Integration Server 2000 libraries.
/D NOTRC	The **NOTRC** macro specifies that internal tracing should not be compiled into the application. See Diagnostics for more information on the use of internal tracing. The **/D NOTRC** option should be used for building a final system (internal tracing should not be included because it will degrade performance and occupancy). For a development system, you may want to compile with internal tracing; if so, remove the **/D NOTRC** option.
/D WIN32_SUPPORT	The macro **WIN32_SUPPORT** is used in the header files SNA_DLC.H, SNA_CNST.H, and TRACE.H to support variants of the DLC interface for Microsoft® Windows 2000/NT/95/98.®
/Gzs	z: Use stdcall conventions (only on i386/i486 processors, not MIPS or Alpha). s: Remove stack check calls.

The following compiler flags are required, but any of the valid options for each flag may be used, as appropriate to your application:

/O	Optimization
/W	Warning level

Linking

The IHVLINK.LIB library must be linked with the application. It contains the Base, DMOD imports, and diagnostics routines. The DMOD and the Base are implemented as DLLs.

Note that the Microsoft® Host Integration Server 2000 or SNA Server library only contain the external references for the corresponding DLLs, which are part of the main Host Integration Server 2000 or SNA Server product.

Synchronous Dumb Card Interface

This section describes the interface to the synchronous dumb card device driver used by the Microsoft® Host Integration Server 2000-supplied SDLC and X.25 SNALinks. The interface provides a simple but flexible mechanism for tranferring frames of data through a dumb synchronous communications card (such as the IBM MPCA card).

This interface is intended primarily for IHVs who wish to provide drivers for their own dumb cards that will be directly compatible with the Host Integration Server 2000-provided SDLC and X.25 SNALinks. It can also be used by IHVs who intend to write their own SNALinks and wish to ensure that their drivers conform to the standard Host Integration Server 2000 model.

The Driver Interface

Application software running on Microsoft® Windows 2000 or Windows NT® normally does not interact directly with device drivers. Usually, the operating system itself controls the interface to underlying device drivers on behalf of the application. For example, Disk I/O consists of sequences of driver requests generated by a file system, as a result of an application making file system requests.

By contrast, in the Microsoft® Host Integration Server 2000 device driver model, synchronous dumb cards are controlled directly by the SNALink using input and output control (IOCTL) commands. This mechanism allows the SNALink to pass raw control packets to the driver without any intervention from the operating system.

This is achieved by issuing an Open request with a file name that identifies the device driver. The operating system detects the fact that this file is in fact a driver and passes an OPEN I/O request packet to the driver. The user application is returned a handle that can be used to reference the driver.

The IBMSYNC driver creates various device names. During setup, the configuration for adapters in the computer is saved in the registry; when the driver starts up, it reads this data and creates the device names for all the adapters that are found.

The following table lists the device names that the IBMSYNC driver can create.

Device name	**Description**
\Device\IBMSDLC	Standard IBM SDLC adapter.
\Device\MPCA_1	IBM MPCA 1 adapter. This adapter has a switch set on it to enforce MPCA 1 operation. This adapter is the primary MPCA adapter in the computer and supports DMA interrupt mode.
\Device\MPCA_2	IBM MPCA 2 adapter. This adapter has a switch set on it to enforce MPCA 2 operation. This adapter is the secondary MPCA adapter in the computer and supports only interrupt mode.
\Device\SYNC_*x*	Generic adapter (for example, Microgate). The letter *x* is 1 (for the primary adapter) or 2 (for the secondary adapter).
\Device\MPAA_S*x*	IBM MPAA adapter, where *x* represents the number of the MCA slot where the adapter is installed in the computer. This number is a value from 1 through 8.
\Device\SYNC_S*x*	Generic MPAA adapter (for example, the Microgate MPAA adapter). The letter *x* represents the number of the MCA slot where the adapter is installed in the computer. This number is a value from 1 through 8.

Subsequent IOCTL calls (using **DeviceIOControl** under Windows NT) made by the SNALink using the driver handle cause the operating system to pass an IOCTL I/O request packet to the driver. The driver therefore sees IOCTL requests from the SNALink as a series of I/O request packets passed to it by the operating system.

The Host Integration Server 2000 dumb card interface uses the following operating system calls:

OpenFile

DeviceIOControl

CloseFile

DeviceIOControl allows free-format information to be passed to the driver. The dumb card interface uses its own format of information to pass all requests to the driver (with the exception of Open and Close requests, which are handled differently by the operating system).

Architecture Overview

This section describes how information is transferred between the driver and the SNALink.

The Interface Record

Status information is transferred between the driver and the SNALink using a buffer known as the interface record.

The driver allocates this buffer when it starts and maintains the information in it while it is running. The contents of this buffer are copied to an SNALink buffer by using an IOCTL call of type **READ_INTERFACE_RECORD**.

Event Signaling

The device driver notifies the SNALink whenever an event occurs (such as a frame being received from the line) by setting an event.

The SNALink provides the driver with a handle to this event (or semaphore) at start of day by issuing an IOCTL call of type **SET_EVENT_HANDLE**.

Link Characteristics

Before the driver can transfer any data, it needs information about the link. This includes:

- The frame size.
- The station address to listen on (if required).
- Details of hardware selectable options, such as SDLC/HDLC, Internal/External clocking, and so on.

For more details of these options, refer to the description of the **SET_LINK_CHARACTERISTICS** IOCTL call.

I/O Request Packets

All I/O requests are passed to the driver by Microsoft® Windows 2000/Windows NT® using the standard IRP structure. For more details of this, refer to the Windows 2000 or Windows NT Device Driver Kit.

I/O request packets are defined in terms of C structures. The relevant fields are accessed as follows:

IRP.CurrentStackLocation -> MajorFunction	Defines the IRP as an IOCTL.
IRP.IoStatus	Status codes upon completion of request.
IRP.CurrentStackLocation -> IoControlCode	The IOCTL function code.

IoControlCode identifies the function to be performed and **IoStatus** is the mechanism for returning result codes to the SNALink. The structure **IOStatus** is defined as follows:

IoStatus.Status
A standard Windows 2000/NT result code (for example, STATUS_INVALID_PARAMETER) as defined in the Windows 2000/NT header file NTSTATUS.H.

IoStatus.Information
For successful read-frame IOCTLs, the length of the received buffer (can be zero if no data available). Additional error information, as defined in the header file SECLINK.H.

Initialization

Device drivers under Windows 2000 or Windows NT should perform all initialization required at start of day when they are loaded by Windows 2000/NT. Configuration information for device drivers is stored in the Configuration Registry under Windows 2000/NT. For more details, refer to the documentation supplied with the Windows 2000 or Windows NT Device Driver Kit.

The SNALinks for dumb cards are implemented by the following files:

SDLC SNALink
SNAroot\SYSTEM\IBMSDLC.DLL
X.25 SNALink
SNAroot\SYSTEM\IBMX25.DLL

To bind to one of these, an installation script should mention the appropriate DLL in the IHVDLL registry entry that the script creates for the link service.

OPEN Call

The **OPEN** call has no parameters. It grants access to the driver from a particular process. The driver ensures that only one **OPEN** is accepted by the link at any one time. When **OPEN** is processed, the driver attempts to reserve access to hardware resources such as interrupt vectors; the **OPEN** is rejected if this fails.

After a successful **OPEN** request, the driver expects to receive the following IOCTL commands:

SET_EVENT_HANDLE

SET_INTERFACE_RECORD

SET_LINK_CHARACTERISTICS

Of these, the first two can be performed in any order, but both should be issued before calling **SET_LINK_CHARACTERISTICS**.

When these three calls have been successfully performed by the SNALink, the driver is ready for information transfer.

CLOSE Call

The **CLOSE** call has no parameters. It performs the logical converse of **OPEN**. Resources are released back to the operating system when a **CLOSE** is performed.

IOCTL Command Summary

The parameters to the IOCTL request packet are stored in the following fields in the associated I/O request packet (IRP).

IRP.CurrentStackLocation -> IOControlCode	**Function code**
IRP.SystemBuffer	Address of parameter buffer (if used)
IRP.CurrentStackLocation -> InputBufferLength	Length of parameter buffer
IRP.UserBuffer	Address of data buffer
IRP.CurrentStackLocation -> OutputBufferLength	Length of data buffer

Note that under Windows 2000 or Windows NT, the operating system reserves the low nibble of IOCTL function codes to determine the method used to map the various buffers passed on the **DeviceIoControl** function call into the driver address space. The various options available to device driver writers are:

Low nibble	IOCTL definition
0	METHOD_BUFFERED
1	METHOD_IN_DIRECT
2	METHOD_OUT_DIRECT
3	METHOD_NEITHER

For further details of the memory mapping performed by these various options, refer to the Windows 2000 or Windows NT DDK documentation.

For a driver function code of ZZ, using memory mapping code M, the IOCTL code passed on the **DeviceIoControl** function call is 0xZZM.

The function codes are set out as shown below. Note that all other function codes will be returned with the error ERROR_INVALID_DEVICE_REQUEST in the field **IoStatus.Status**. The Windows 2000/NT I/O System validates the address and length of the areas passed as parameter and data packets. If the address validation fails, an exception will be raised.

All requests must return immediately. In general, they are simple, immediate operations, but in the case of Transmit Frame and Receive Frame, the driver must not suspend the calling SNALink thread while waiting for I/O to complete — a relevant return code should be sent instead, allowing the SNALink to retry.

The complete list of functions is as follows:

Function	Function code	Windows 2000/NT IOCTL code
Set Event/Semaphore Handle	0x41	0x410
Set Link Characteristics	0x42	0x420
Set V24 Output Status	0x43	0x430
Transmit Frame	0x44	0x441
Abort Transmitter	0x45	0x450
Abort Receiver	0x46	0x460
Off-Board Load	0x47	0x470
Get/Set Interface Record	0x61	0x613
Get V24 Status	0x62	0x622
Receive Frame	0x63	0x632
Read Interface Record	0x64	0x642

In the function descriptions in the following topics, the bit-numbering convention is: The bits in a byte are numbered 0 through 7, where bit 0 is the least significant and bit 7 is the most significant.

> There is no function for the controlling autodialer across the synchronous dumb card interface. This autodial feature is implemented in the link service itself. The Microsoft link services that support the synchronous dumb card interface first perform the dial operation by sending the dial string containing the server-stored number to a COM port rather than the SDLC chip, and then sending a command to the device driver to raise DTR via an Set V24 Output Status IOCTL.

Equates and Structure Layouts

Many standard operating system device driver error codes are used (for example, "invalid function"), together with a new set of device driver-specific errors. Return codes below 0x80 reflect serious failures.

```
/* Copyright Data Connection Ltd. 1989 */_
/*****************************************************************************/
/* Link Device Driver interface constants and structures. */
/*****************************************************************************/
/*****************************************************************************/
/* WIN32  16/04/92  SW   Added more helpful names from WIN32 hdr file        */
/* IHV    03/06/92  MF2  Add semfisui.h                                      */
/*****************************************************************************/

/*****************************************************************************/
/* This include file is used in 5 subsystems                                 */
/*                                                                           */
/* - the NT   driver                          LINK_NTDRIVER     */
/* - the X25  link service for NT             LINK_NTX25        */
/* - the SDLC link service for NT             LINK_NTSDLC       */
/* - the X25  link service for OS/2           LINK_OS2X25       */
/* - the SDLC link service for OS/2           LINK_OS2SDLC      */
/*                                                                           */
/* (The OS/2 driver doesn't count because it is in assembler). */
/*                                                                           */
/* These are distinguished by #defines as set in the following */
/*                                                                           */
/*****************************************************************************/

#ifdef        IMADRIVER
  #define     LINK_NTDRIVER
#else
  #ifdef      SDLC
    #ifdef    WIN32
      #define LINK_NTSDLC
    #else
      #define LINK_OS2SDLC
    #endif
  #else
    #ifdef    WIN32
      #define LINK_NTX25
    #else
      #define LINK_OS2X25
    #endif
  #endif
#endif
/*****************************************************************************/
/* Device function codes for DosDevIOCtl to link device driver               */
/*****************************************************************************/
#ifdef WIN32                /* WIN32 constants defined in semfisui.h         */
#define IoctlCodeSetEvent               0x410
#define IoctlCodeSetLinkChar            0x420
#define IoctlCodeSetV24                 0x430
#define IoctlCodeTxFrame                0x440
#define IoctlCodeAbortTransmit          0x450
#define IoctlCodeAbortReceiver          0x460
#define IoctlCodeSetInterfaceRecord     0x610 /*IRMdl?*/
#define IoctlCodeGetV24                 0x623
#define IoctlCodeRxFrame                0x633
#define IoctlCodeReadInterfaceRecord    0x643   /*IRMdl?*/
#else
//obsolete names from previous version
//#define CELDDSSH 0x41       /* Set Semaphore Handle                          */
//#define CELDDSLC 0x42       /* Set Link Characteristics                      */
//#define CELDDSVS 0x43       /* Set V24 Output status                         */
//#define CELDDTXF 0x44       /* Transmit a frame of data                      */
```

```
//#define CELDDATX 0x45       /* Abort Transmitter                          */
//#define CELDDARX 0x46       /* Abort Receiver                             */
//#define CELDDGIR 0x61       /* Get Interface Record Address               */
//#define CELDDGVS 0x62       /* Get V24 Input Status                       */
//#define CELDDRXF 0x63       /* Receive a frame of data                    */
//#define CELDDCAT 0x82       /* Device function category code              */
//
// new names

#define IoctlCodeSetEvent               0x41
#define IoctlCodeSetLinkChar            0x42
#define IoctlCodeSetV24                 0x43
#define IoctlCodeTxFrame                0x44
#define IoctlCodeAbortTransmit          0x45
#define IoctlCodeAbortReceiver          0x46
#define IoctlCodeSetInterfaceRecord     0x61
#define IoctlCodeGetV24                 0x62
#define IoctlCodeRxFrame                0x63

#endif

/*****************************************************************************/
/* Constants for the driver-specific IOCtl return codes.                     */
/*****************************************************************************/
#define CEDNODMA 0xff80     /* Warning (NO DMA!) from set link chrctrstcs    */
/*****************************************************************************/
/* Equates for the link options byte 1.                                      */
/*****************************************************************************/
#define CEL4WIRE 0x80
#define CELNRZI  0x40
#define CELPDPLX 0x20
#define CELSDPLX 0x10
#define CELCLOCK 0x08
#define CELDSRS  0x04
#define CELSTNBY 0x02
#define CELDMA   0x01

/*****************************************************************************/
/* Equates for the driver set link characteristics byte 1.                   */
/*****************************************************************************/
#define CED4WIRE 0x80
#define CEDNRZI  0x40
#define CEDHDLC  0x20
#define CEDFDPLX 0x10
#define CEDCLOCK 0x08
#define CEDDMA   0x04
#define CEDRSTAT 0x02
#define CEDCSTAT 0x01

/* Nicer names for NT-style code */

#define LinkOption_4Wire           CED4WIRE
#define LinkOption_NRZI            CEDNRZI
#define LinkOption_HDLC            CEDHDLC
#define LinkOption_FullDuplex      CEDFDPLX
#define LinkOption_InternalClock   CEDCLOCK
#define LinkOption_DMA             CEDDMA
#define LinkOption_ResetStatistics CEDRSTAT

/*****************************************************************************/
/* Equates for the output V24 interface flags.                               */
/*****************************************************************************/
#define CED24RTS 0x01
#define CED24DTR 0x02
#define CED24DRS 0x04
#define CED24SLS 0x08
#define CED24TST 0x10
```

```
/* Nicer names for NT-style code */

#define IR_OV24RTS  CED24RTS
#define IR_OV24DTR  CED24DTR
#define IR_OV24DSRS CED24DRS
#define IR_OV24SlSt CED24SLS
#define IR_OV24Test CED24TST


/*************************************************************************/
/* Equates for the input V24 interface flags.                            */
/*************************************************************************/
#define CED24CTS 0x01
#define CED24DSR 0x02
#define CED24DCD 0x04
#define CED24RI  0x08

/* Nicer names for NT-style code */

#define IR_IV24CTS  CED24CTS
#define IR_IV24DSR  CED24DSR
#define IR_IV24DCD  CED24DCD
#define IR_IV24RI   CED24RI
#define IR_IV24Test 0x10


/*************************************************************************/
/* Structure for the device driver interface record.                     */
/*************************************************************************/

#define CEDSTCRC  0         /* Frames received with incorrect CRC            */
#define CEDSTOFL  1         /* Frames received longer than the maximum       */
#define CEDSTUFL  2         /* Frames received less than 4 octets long       */
#define CEDSTSPR  3         /* Frames received ending on a non-octet bndry   */
#define CEDSTABT  4         /* Aborted frames received                       */
#define CEDSTTXU  5         /* Transmitter interrupt underruns               */
#define CEDSTRXO  6         /* Receiver    interrupt overruns                */
#define CEDSTDCD  7         /* DCD (RLSD) lost during frame reception        */
#define CEDSTCTS  8         /* CTS lost while transmitting                   */
#define CEDSTDSR  9         /* DSR drops                                     */
#define CEDSTHDW 10         /* Hardware failures - adapter errors            */

#define CEDSTMAX 11

#define SA_CRC_Error       CEDSTCRC
#define SA_RxFrameTooBig   CEDSTOFL
#define SA_RxFrameTooShort CEDSTUFL
#define SA_Spare           CEDSTSPR
#define SA_RxAbort         CEDSTABT
#define SA_TxUnderrun      CEDSTTXU
#define SA_RxOverrun       CEDSTRXO
#define SA_DCDDrop         CEDSTDCD
#define SA_CTSDrop         CEDSTCTS
#define SA_DSRDrop         CEDSTDSR
#define SA_HardwareError   CEDSTHDW  /* e.g. CmdBufferFull not set  */

#define SA_Max_Stat        CEDSTMAX

#ifdef WIN32

typedef struct         _INTERFACE_RECORD
{
  int                   RxFrameCount;   /* incremented after each frame rx'd */
  int                   TxMaxFrSizeNow; /* max available frame size av. now  */
                                        /* (changes after each Tx DevIoctl   */
                                        /* to DD or after Tx completed)      */
  int                   StatusCount;    /* How many status events have been  */
                                        /* triggered.                        */
```

```
  UCHAR                 V24In;          /* Last 'getv24i/f' value got        */
  UCHAR                 V24Out;         /* Last 'setv24 outputs' value set   */


/*       The values for the indexes into the link statistics array of the */
/*       various types of statistic. */

  int                   StatusArray[SA_Max_Stat];

}                       IR,
                     * PIR;

#else
typedef struct teifrec {

        USHORT    RxFrameCount;
        USHORT    TxMaxFrSizeNow;
        USHORT    StatusCount;
        UCHAR     V24In;
        UCHAR     V24Out;
        USHORT    StatusArray[CEDSTMAX];

        }TEIFREC;

typedef TEIFREC far * TEIFRPTR;
#endif

/*****************************************************************************/
/* Structure for the set link characteristics parameter block.               */
/*****************************************************************************/

#ifdef WIN32
typedef struct  _SLPARMS
{
        int     SLFrameSize;            /* max frame size on link - must be  */
                                        /* in range 270 to ?2K-ish           */
        LONG    SLDataRate;             /* not used by us - external clocks  */
        UCHAR   SLOurAddress1;          /* ) e.g C1/FF or 00/00 or 01/03     */
        UCHAR   SLOurAddress2;          /* )                                 */
        UCHAR   SLLinkOptionsByte;      /* see documentation & LinkOption_*  */
        UCHAR   SLSpare1;
}
                SLPARMS;
#else

typedef struct teslcrec {

        USHORT  SLFrameSize;
        ULONG   SLDataRate;
        UCHAR   SLOurAddress1;
        UCHAR   SLOurAddress2;
        UCHAR   SLLinkOptionsByte;
        UCHAR   SLSpare1;

        }TESLCREC;

#endif

/*****************************************************************************/
/* DEVICEIOCTL macros                                                        */
/*****************************************************************************/
#ifdef WIN32
/* NT_SUCCESS ripped of from DDK's ntdef.h, which we do not want to include  */
/* for now temporarily (12/5/92)                                             */
#define NT_SUCCESS(Status) ((NTSTATUS)(Status) >= 0)

#define SETEVENTHANDLE(H) NtDeviceIoControlFile(                  \
                seldrvrh,                   \
```

```
                H,                          \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                &IoStatus,                  \
                IoctlCodeSetEvent,          \
                ( PVOID ) NULL,             \
                0L,                         \
                ( PVOID ) NULL,             \
                0L                          \
               )

#define SETINTERFACERECORD(R) NtDeviceIoControlFile(   \
                seldrvrh,                   \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                &IoStatus,                  \
                IoctlCodeSetInterfaceRecord,           \
                                            \
                &R,                         \
                sizeof(R),                  \
                ( PVOID ) NULL,             \
                0L                          \
               )

#define SETV24STATUS NtDeviceIoControlFile(            \
                seldrvrh,                   \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                &IoStatus,                  \
                IoctlCodeSetV24,            \
                NULL,             \
                0L,                         \
                &pInterfaceRecord->V24Out,  \
                1L                          \
               )


/*****************************************************************************/
/* The above change is temporary!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
/*****************************************************************************/

#define GETV24STATUS NtDeviceIoControlFile(   \
                  seldrvrh,                 \
                  ( PVOID ) NULL,           \
                  ( PVOID ) NULL,           \
                  ( PVOID ) NULL,           \
                  &IoStatus,                \
                  IoctlCodeGetV24,          \
                  ( PVOID ) NULL,           \
                  0L,                       \
                  ( PVOID ) NULL,           \
                  0L                        \
                 )

#define SETLINKCHARACTERISTICS(A) NtDeviceIoControlFile(      \
                seldrvrh,                   \
                NULL,                       \
                ( PVOID ) NULL,             \
                ( PVOID ) NULL,             \
                &IoStatus,                  \
                IoctlCodeSetLinkChar,       \
                &A,                         \
                sizeof(A),                  \
                ( PVOID ) NULL,             \
                0L                          \
               )
```

```
#define TRANSMITFRAME(A,B) NtDeviceIoControlFile(            \
                 seldrvrh,                 \
                 NULL,                     \
                 ( PVOID ) NULL,           \
                 ( PVOID ) NULL,           \
                 &IoStatus,                \
                 IoctlCodeTxFrame,         \
                 ( PVOID ) NULL,           \
                 0L,                       \
                 A,                        \
                 B                         \
                )

#define RECEIVEFRAME(A,B) NtDeviceIoControlFile(             \
                 seldrvrh,             \
                 NULL,                 \
                 ( PVOID ) NULL,       \
                 ( PVOID ) NULL,       \
                 &IoStatus,            \
                 IoctlCodeRxFrame,     \
                 ( PVOID ) NULL,       \
                 0L,                   \
                 A,                    \
                 B                     \
                )

#define READINTERFACERECORD NtDeviceIoControlFile( \
                 seldrvrh,             \
                 NULL,                 \
                 ( PVOID ) NULL,       \
                 ( PVOID ) NULL,       \
                 &IoStatus,            \
                 IoctlCodeReadInterfaceRecord, \
                 ( PVOID ) NULL,       \
                 0L,                   \
                 &InterfaceRecord,     \
                 sizeof(InterfaceRecord) \
                )

#else
#define NT_SUCCESS(R) ((R) == 0)

#define SETEVENTHANDLE(H)  DosDevIOCtl(NULL,           \
                   &H,                         \
                   IoctlCodeSetEvent,          \
                   CELDDCAT,                   \
                   seldrvrh)

#define GETINTERFACERECORD(P) DosDevIOCtl(NULL,        \
                   &P,                         \
                   IoctlCodeGetInterfaceRecord,        \
                   CELDDCAT,                   \
                   seldrvrh)

#define SETV24STATUS DosDevIOCtl(NULL,                 \
                   NULL,                       \
                   IoctlCodeSetV24,            \
                   CELDDCAT,                   \
                   seldrvrh)

#define GETV24STATUS DosDevIOCtl(NULL,                 \
                     NULL,                     \
                     IoctlCodeGetV24,          \
                     CELDDCAT,                 \
                     seldrvrh)

#define SETLINKCHARACTERISTICS(A) DosDevIOCtl((long) NULL,  \
```

```
                    (char far *) &A,                        \
                    IoctlCodeSetLinkChar,                   \
                    CELDDCAT,                               \
                    seldrvrh)

#define TRANSMITFRAME(F,L) DosDevIOCtl(F,               \
                    &L,                                 \
                    IoctlCodeTxFrame,                   \
                    CELDDCAT,                           \
                    seldrvrh)

#define RECEIVEFRAME(F,L)  DosDevIOCtl(F,               \
                           &L,                          \
                           IoctlCodeRxFrame,            \
                           CELDDCAT,                    \
                           seldrvrh)
#endif

//############################################################################

/*************************************************************************/
/* INFO_ : additional information error codes put in IoStatus.Information    */
/*************************************************************************/

#define INFO_CANT_ALLOCATE_SPINLOCK      1
#define INFO_CANT_CONNECT_INTERRUPT      2
#define INFO_HARDWARE_INIT_FAILURE       3
#define INFO_SET_EVENT_NO_EVENT          4
#define INFO_HARDWARE_CMD_TIMEOUT        5
#define INFO_LINKCHAR_BUF_WRONG_SIZE     6
#define INFO_FRAME_BUF_TOO_BIG           7
#define INFO_FRAME_BUF_TOO_SMALL         8
#define INFO_NO_CLOCKS                   9
#define INFO_NO_DMA_FDX                 10
#define INFO_CANT_ALLOCATE_MDL          11
#define INFO_CANT_ALLOCATE_MEMORY       12
#define INFO_DMA_BUFFER_UNUSABLE        14
#define INFO_TX_BUFFER_FULL             15
#define INFO_TX_FRAME_TOO_BIG           16
#define INFO_TX_FRAME_TOO_SMALL         17
#define INFO_READ_IR_BUFFER_WRONG_SIZE  18
#define INFO_NEEDS_MCA_BUS              19
#define INFO_NEEDS_ISA_BUS              20
```

SNA Modem Status Interface

This section describes the interface to the modem status used by the Microsoft® Host Integration Server 2000-supplied SDLC and X.25 SNALinks. This interface provides a set of RASmon-like modem lights. The modem status interface is intended primarily for the Microgate cards with internal modems, but can be used with any SDLC/X.25 link service to show the modem status.

This interface is intended primarily for IHVs who wish to provide drivers for their own dumb cards that will be directly compatible with the Host Integration Server 2000-provided SDLC and X.25 SNALinks. It can also be used by IHVs who intend to write their own SNALinks but who wish to ensure that their drivers conform to the standard Host Integration Server 2000 model.

SNA Device Driver Interface to Modem Status

There are three interfaces from which modem status can be gathered:

- The SNADIS Dumb Card Interface **GetV24Status** IOCTL call that reports the status of the Ring Indicator (DRI), Carrier Detect (DCD), Clear To Send (CTS), and Data Set Ready (DSR) signal lines.
- The SNADIS Dumb Card Interface **SetV24Status** IOCTL call that sets the status of the Data Terminal Ready (DTR) and Request To Send (RTS) signal lines.
- The SNADIS Dumb Card Interface receive and transmit IOCTL calls, used to set the operating mode of the device driver.

Whenever one of the first two interfaces indicates a modem line is high, the corresponding light in the display is lit. However, the transmit and receive IOCTL calls cannot provide a definitive statement as to whether the card is receiving data, only that it is ready to receive data. To work around this limitation and to have the modem lights give a reasonable semblance of ordinary data throughput, the following mechanism is recommended.

The receive and transmit lights are simulated by counters. Each link service tracks the number of frames received and transmitted. The display application interpolates the flashing of the receive and transmit lights from these counters.

The remainder of this section describes the changes that must be made to the link service for supporting the modem status interface and the interface provided to the display application. Microsoft® Host Integration Server 2000 comes with an application that displays the modem status.

Supporting Modem Status in an SNA Link Service

The implementation of a link service that supports the modem status requires the following:

- A call to the SNA Modem interface to initialize the SNA Modem data structures.
- Modification of the SNA Modem structures as the status changes, as reported by the DevIoctl calls.

IHVs who use the Microsoft® Host Integration Server 2000 SDLC link service and who fully implement the SNADIS interface will have no changes to make to use the modem status feature. However, IHVs should check that the status returned by DevIoctl calls from their device driver conforms to the requirements described below.

IHVs who provide both the device driver and link service need to implement the code in their link service that initializes the SNA Modem API and updates the modem status information.

Modem API Summary

To simplify the task for IHVs who want to use this feature, four new entry points have been added to SNALINK.DLL. An IHV who uses these must be linking with IHVLINK.LIB, a stub library that contains the exports library for SNALINK.DLL. This API allows the IHV to simply maintain the contents of a MODEM_STATUS structure. The underlying SNALINK library code handles the communication of this information to the modem lights application.

The modem status functions are as follows:

Function	Description
SNAModemInitialize	Initializes the communication path to the SNA Modem application.
SNAModemAddLink	Adds an SNA link.
SNAModemDeleteLink	Deletes the resources associated with a link.
SNAModemTerminate	Terminates an SNA link.

DevIoctl Definitions to Support SNA Modem Status

The SNA DevIoctl interface is modified to update the MODEM_STATUS structure for a link each time a modem status change is detected or caused by a **GetV24** or **SetV24** IOCTL call. Code is manually added to the link service to track the number of frames received and transmitted.

The DevIoctl changes are highlighted below:

```
#define SETV24STATUS                                                \
         NtDeviceIoControlFile(seldrvrh,NULL,NULL,NULL,&IoStatus, \
                               IoctlCodeSetV24,NULL,0L,           \
                               &pInterfaceRecord->V24Out,1L);     \
         if (SavedIROut != (InterfaceRecord.V24Out &              \
                            (MASK_DTR | MASK_RTS)    ))          \
         {                                                        \
           SavedIROut = (pInterfaceRecord->V24Out &               \
                         (MASK_DTR | MASK_RTS)    );              \
           pSharedMem->V24Out = pInterfaceRecord->V24Out;         \
         }


#define GETV24STATUS(rc)                                            \
         rc |= NtDeviceIoControlFile(seldrvrh,NULL,NULL,NULL,     \
             &IoStatus,IoctlCodeGetV24,NULL,0L,NULL,0L);          \
         rc |= READINTERFACERECORD;                               \
         if (SavedIRIn != (InterfaceRecord.V24In &                \
                     (MASK_CTS | MASK_DSR | MASK_DCD| MASK_DRI))) \
         {                                                        \
             SavedIRIn = (InterfaceRecord.V24In &                 \
                     (MASK_CTS | MASK_DSR | MASK_DCD| MASK_DRI)); \
             pSharedMem->V24In = InterfaceRecord.V24In;           \
         }
```

pSharedMem is a pointer to the MODEM_STATUS structure for this link service.

SavedV24In and *SavedV24Out* are characters used to only notify the display application when status changes, not each time it is read or set.

SNA Performance Monitor Interface

This section describes the interface for performance monitoring (Perfmon) used by the Microsoft® Host Integration Server 2000-supplied SNADIS links. This interface is provided to simplify the integration of SNADIS-compliant link services with the Microsoft® Windows 2000 System Monitor and Windows NT® Performance Monitor applications. It provides a common look and feel to all link service performance counters exported by SNADIS links, independent of the vendor and link transport (channel, twinax, SDLC, X.25, TR, E/Net, and so on).

The performance monitoring statistics maintained for an SNA link service are stored in a series of ADAPTERCOUNTER structures that are members of an ADAPTERPERFDATA structure. These structures are defined in the SEMFPERF.H header file.

Three API entry points are exported from IHVLINK.DLL (and the IHVLINK.LIB import library) that are used by the Perfmon API. These functions should be called in the order noted below at link service initialization time.

To support perfomance monitoring, an SNA Link driver first calls SNAInitLinkPerfMon to initialize data structures used by the Perfmon application. This call should be followed with a call to function SNAGetLinkPerfArea, which returns a shared mutex handle and a pointer to the shared data area for the **ADAPTERPERFDATA** structure used by the Perfmon application to store the link statistics. This handle and shared memory data area parameter are the returned values from **SNAInitLinkPerfmon**. Finally, the SNAGetPerfValues function is called to fill in the *ServiceNameIndex* and *FirstCounterIndex* fields so that the Perfmon application knows where to get the descriptions of the performance counters from the registry.

After these three calls have been made, the SNA link driver simply maintains the count members in the **ADAPTERCOUNTER** structures that make up the **ADAPTERPERFDATA** structure, incrementing the count member whenever data is received, connections fail, and other events occur. The Perfmon application accesses these counters to display Microsoft® Host Integration Server 2000 performance monitoring data statistics.

SNADIS Reference

This section provides reference material for developers writing their own SNALink software.

This section contains:

Base/DMOD and SNALink Entry Points

This section gives definitions for Base/DMOD and SNALink entry points that must be supplied in an SNALink.

SNAGetBuffer

The **SNAGetBuffer** function is called by an application to get a buffer with a requested number of elements.

```
PTRBFHDR SNAGetBuffer(
INTEGER numelts
);
```

Parameters

numelts
Number of elements required.

Return Values

A pointer to the buffer obtained; NULL if a buffer could not be obtained.

Remarks

Each element has a size of 268, the constant SNANBEDA in the header file SNA_DLC.H.

The returned buffer consists of a header and the required number of elements. The header points to the first element, which points to the next element and so on to make an element chain.

It is possible to add an element to an existing buffer by calling SNAGetElement to get the extra element. The new element should be added to the element chain of the buffer, and the number of elements count should be updated.

The application must release any buffers that are not transmitted.

SNAGetElement

The **SNAGetElement** function is called by an application to get a buffer element to append to an existing buffer.

```
VOID SNAGetElement(
PTRBFELT *eltptr
);
```

Parameters

eltptr
Pointer to a pointer to an element. On return this points to a pointer to the element obtained, or to NULL if an element was not obtained (an internal error).

Remarks

This function should only be used to get extra elements for an existing buffer. SNAGetBuffer should be used to get a new buffer.

The new element should be added to the chain of elements from the existing buffer header and the count of the number of elements updated.

This function is typically used when a received buffer is being reused to transmit a message that is longer than the incoming message.

SNAGetLinkName

The SNALink can call the **SNAGetLinkName** function to obtain its configured SNALink name.

```
VOID SNAGetLinkName(
UCHAR *linkname
);
```

Parameters

linkname
A pointer to a buffer where the NULL-terminated SNALink name is stored.

Remarks

The buffer should be at least nine bytes in length.

SNALinkDispatchProc

The **SNALinkDispatchProc** function is the link dispatcher function. The Base calls this function whenever one of the following events occurs:

- A message arrives for the link.
- The Base timer expires.
- Contact is lost with the local node.

```
VOID SNALinkDispatchProc(
PTRBFHDR msgptr,
INTEGER function,
INTEGER locality
);
```

Parameters

msgptr
The message to be processed, or NULL if some other event is being notified.
function
The reason for **SNALinkDispatchProc** being called.
locality
L value (only valid for function SBLOST).

Remarks

The *function* parameter can have one of three values:

- 0—Message received.
- SBLOST—Contact lost with local node; L-value of locality.
- SBTICK—Base timer has expired; occurs every five seconds.

See Sample Code for SNALinkDispatchProc for suggested usage of this function.

SNALinkInitialize

The **SNALinkInitialize** function initializes the SNALink. The Base calls this function when the SNALink is loaded into memory.

```
VOID SNALinkInitialize(
HANDLE event
);
```

Parameters

event
A handle to the global Base event.

Remarks

This function should:

- Read in required configuration information.
- Perform any required initialization of the hardware or device driver.
- Set up control blocks and data structures required internally by the SNALink.

SNALinkTerminate

The **SNALinkTerminate** function terminates the SNALink.The Base calls this function, when present, during service shutdown. This allows the DLL to free memory, release system resources (such as events), and close drivers.

```
VOID SNALinkTerminate(void);
```

Remarks

This function must not send messages to other SNA components.

SNALinkWorkProc

The **SNALinkWorkProc** function is the work manager function. The Base calls this function whenever the global Base event is triggered by the SNALink, or at least once every five seconds.

```
VOID SNALinkWorkProc(void);
```

Remarks

This function can be used to perform any general processing required by the SNALink, in particular to process messages received from the link.

SNAReleaseBuffer

The **SNAReleaseBuffer** function is called by an application to release a buffer.

```
VOID SNAReleaseBuffer(
PTRBFHDR msgptr
);
```

Parameters

msgptr
Pointer to the buffer to be released.

Remarks

It is important that buffers are released after use. This is done automatically when a message is transmitted. For messages received, it is the responsibility of the application either to release or to reuse the buffer.

This function releases both the buffer header and any associated buffer elements. It is possible to release single elements from a buffer by using the function SNAReleaseElement.

SNAReleaseElement

The **SNAReleaseElement** function is called by an application to release a buffer element from an existing buffer.

```
VOID SNAReleaseElement(
PTRBFELT *eltptr
);
```

Parameters

eltptr
Pointer to a pointer to the element to be released.

Remarks

This function should only be used to release surplus elements from a buffer. SNAReleaseBuffer should be called to release the entire buffer.

The released element should first be removed from the element chain and the count of the number of elements updated.

This function is typically used when a received buffer is being reused to transmit a message that is shorter than the incoming message.

SNASendAlert

The SNALink calls the **SNASendAlert** function to send a complete preformatted Network Management Vector Transport (NMVT) alert to NetView.

```
VOID SNASendAlert(
PTRBFHDR msgptr,
INTEGER severity
);
```

Parameters

msgptr
Pointer to the NMVT alert to be sent.
severity
The severity of the problem that caused the alert (ranges from 0 to 16).

Remarks

The complete NMVT to be sent must be generated by the SNALink and inserted into a buffer. Only the elements are used—the buffer header need not be set up before sending. The fields **startd** and **endd** should be set to reflect the location of the NMVT within the element. Multiple elements can be used to store the NMVT, up to a maximum length of 512 bytes. The buffer will be freed by the Base after the NMVT has been sent.

Any NMVT sent refers to a particular Host Integration Server 2000 connection. It is recommended that the NMVT include at least a hierarchy resource list, giving the name of the remote PU the connection is associated with. This name is supplied to the SNALink on the Open(STATION) message.

For complete details of the format of an NMVT, see the IBM manual *SNA Formats* (GA27-3136).

SNASendMessage

The **SNASendMessage** function is called by an application to send messages to other localities (in the case of an SNALink, the local 2.1 node).

```
VOID SNASendMessage(
PTRBFHDR *msgptr
);
```

Parameters

msgptr
Pointer to message to be sent.

Remarks

The LPI values on the message should be set up to reference the correct connection for the data to be passed.

Message Formats

This section describes the SNA Device Interface Specification interface in terms of message formats. These are presented in a language-independent notation that is described below.

The messages used between the node and the SNALinks are shown in the following table.

Message type	Direction	LPI connection
Open(LINK)Request	NODE ------> DLC	LINK
Close(LINK)Request	NODE ------> DLC	LINK
Send-XID	NODE ------> DLC	LINK
Open(STATION) Request	NODE ------> DLC	STATION
Close(STATION) Request	NODE ------> DLC	STATION
Open(LINK) Response	NODE <------ DLC	LINK
Close(LINK) Response	NODE <------ DLC	LINK
Request-Open-Station	NODE <------ DLC	LINK
Open(STATION) Response	NODE <------ DLC	STATION
Close(STATION) Response	NODE <------ DLC	STATION
Station-Contacted	NODE <------ DLC	STATION
Outage	NODE <------ DLC	LINK/STATION
DLC-Data	NODE <-----> DLC	STATION
Status-Resource	NODE <-----> DLC	STATION

Details of the message format notation and key assumptions about the contents of the message formats are as follows:

- "Reserved" indicates that the field must be set to zero (for a numeric field) or all nulls (for names) by the sender of the message.
- "Undefined" indicates that the value of the field is indeterminate. The field is not set by the sender and should not be examined by the receiver of the message.
- Fields that occupy two bytes—the **srci** field in all messages, and fields such as **opresid** in Open(LINK) Request —are represented with the arithmetically most significant byte in the lowest byte address, irrespective of the normal byte order used by the processor on which the software executes. That is, the 2-byte value 0x1234 has the byte 0x12 in the lowest byte address. The exception to this is the **startd** and **endd** fields in all elements, which are always stored in the processor's normal byte order.
- Messages are composed of buffers, consisting of a buffer header and zero or more buffer elements; see Messages for more information on buffer formats.
- The **startd** field in each element gives the offset of the first byte of data in the element after the **trpad** field. Its value will either be 1 (data starts in the byte after the **trpad** field), 10 (nine bytes of padding are included between the **trpad** field and the start of the data), or 13 (12 bytes of padding are included between the **trpad** field and the start of the data). Any extra bytes are used by the local node for additional header information. This avoids having to copy data into a new buffer when adding this information.
- Because **startd** indicates the index into **dataru** starting from 1, not 0, the first byte of valid data will always be at **dataru[startd–1]**.
- All fields within **dataru** are of type unsigned character (UCHAR), except where the notes indicate otherwise.

Open(LINK)

Open(LINK) is used by the node to open the LINK LPI connection.

Open(LINK) Request

Flow : NODE ------> DLC

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements: 1 (Number of elements can be 2 if the connection is for an X.25 SVC)
msgtype	CHAR	Message type: OPENMSG (0x01)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
ophdr.openqual	CHAR	Open qualifier: REQU (0x01)
ophdr.opentype	CHAR	Open type: LINK (0x10)
ophdr.opresid	INTEGER	Resource identifier

Element 1

Field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to optional second buffer element (NULL if only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+9]		Source name—name of local node
dataru[s+10..s+19]		Destination name—name of remote PU (blank for incoming calls)
dataru[s+20..s+21]		Link index

Link Data (depends on DLC type)

Unless otherwise stated, these fields are valid for outgoing calls only.

SDLC link data field	Description
dataru[s+22]	XID supplied 0x00 Do not send initial XID 0x01 Send initial XID from this message (may be a NULL XID)
dataru[s+23]	Link operational role 0x00 Primary 0x01 Secondary 0x02 Negotiable
dataru[s+24]	Use Reject_Command indicator 0x00 Do use it 0x01 Do not use it
dataru[s+25]	Address match byte 0x00 Primary/Negotiable SDLC 0x01 to 0xFE Secondary SDLC
dataru[s+26]	Second SDLC address match byte 0x00 Primary SDLC 0xFF Secondary/Negotiable SDLC
dataru[s+27]	Reserved

Channel adapter link data	Description
dataru[s+22]	XID supplied 0x00 Do not send initial XID 0x01 Send initial XID from this message (may be a NULL XID)

dataru[s+23]	PU emulation type 0x00 Unknown 0x20 PU 2.0 (format 0 XID) 0x21 PU 2.1 (format 3 XID)
dataru[s+24]	Control Unit Image number (0 to 15) on the Channel address configured in SNA Manager
dataru[s+25]	Channel subaddress
dataru[s+26..s+67]	Reserved (may not be zero)

Station timers	**Description**
dataru[s+28..s+29]	Contact time-out
dataru[s+30..s+31]	Contact retry limit
dataru[s+32..s+33]	Discontact time-out
dataru[s+34..s+35]	Discontact retry limit
dataru[s+36..s+37]	Negative poll time-out
dataru[s+38..s+39]	Negative poll retry limit
dataru[s+40..s+41]	T1 (no acknowledgment) time-out
dataru[s+42..s+43]	T2 (acknowledgment) time-out
dataru[s+44..s+45]	Remote station busy time-out
dataru[s+46..s+47]	Remote station busy retry limit

Link timers	**Description**
dataru[s+48..s+49]	Idle time-out
dataru[s+50..s+51]	Idle retry limit
dataru[s+52..s+53]	Nonproductive receive time-out
dataru[s+54..s+55]	Nonproductive receive retry limit
dataru[s+56..s+57]	Write time-out
dataru[s+58..s+59]	Write retry limit
dataru[s+60..s+61]	Link connection time-out
dataru[s+62..s+63]	Link connection retry limit 0xFFFF for infinite retry
dataru[s+64..s+65]	Reserved
dataru[s+66]	Configuration options: Bit 0 : 1 = Constant carrier selected Bit 1 : 1 = NRZI 0 = NRZ Bit 2 : = Reserved Bit 3 : 1 = Full-duplex 0 = Half-duplex Bit 4 : 0 = External clocking Bit 5 : 1 = Data signal rate select high 0 = Data signal rate select low Bit 6 : 1 = Select standby on 0 = Select standby off Bit 7 : = Reserved
dataru[s+67]	Configuration options: line type 0x00 leased 0x01 switched manual dial 0x02 switched auto-dial

Note that for configuration options, bit 0 is the most significant option and bit 7 is the least significant. Reserved bits are not always zero, so always use a bitwise **AND** operation when testing these bits.

The PU emulation type is returned as 0x00 (unknown) for versions of SNACFG.DLL supplied with versions of Microsoft® SNA Server earlier than 3.0.

The configuration options byte is also valid for incoming calls.

Expanded Information About Message Formats for Open(LINK) Request with SDLC

The following list supplements the information found in the table in Open(LINK) Request. The timers described in the lists are used by an SDLC link service to determine when to retry communication and when to generate outages. Generally, after the time interval specified by the time-out (usually 1000 milliseconds), the communication is retried. The cycle of time-out and retry is repeated until the retry limit is reached. Then an Outage message is sent by the SDLC link service.

With some timers, there are no communication retries. Such timers simply cycle through the time-out as many times as allowed in the retry limit (without actually retrying), then generate an **Outage** message.

Each description indicates whether you can configure the field through an Host Integration Server 2000 interface (such as the SNA Manager program). If the field is not configurable, the built-in setting for the field is shown.

For information about configuring with SNA Manager, see the Host Integration Server 2000 documentation or the SNA Manager Help.

SDLC Link Data

dataru[s+22] XID supplied

This field controls whether an initial XID is sent on this connection. The value used is determined by the leased/switched setting for the line:

Leased line: 0x00 Do not send an initial XID
Switched line: 0x01 Send an initial XID (may be a NULL XID)

A line is configured as leased or switched in Host Integration Server 2000 Setup.

dataru[s+24] Use Reject_Command indicator

This field determines that the link service will not send a Reject command (an SDLC command, not often used, value 0x19) if a frame is received with an invalid NS (next-to-send) value. Instead, the link service waits until the next poll before requesting retransmission of the frame.

This field is not configurable and must remain at the setting of "Do not use."

Station Timers (described for SDLC only)

dataru[s+28..s+29] Contact time-out
dataru[s+30..s+31] Contact retry limit

This timer is started when an XID or set normal response mode (SNRM) is transmitted. If the time-out expires without acknowledgment, the frame is retransmitted. When the number of retransmitted frames reaches the retry limit, an outage is generated. Note that for XIDs, the time-out value is randomized to prevent possible clashes between two servers sending XIDs simultaneously.

This timer is configurable in SNA Manager, in the advanced settings for an SDLC connection.

dataru[s+32..s+33] Discontact time-out
dataru[s+34..s+35] Discontact retry limit

This timer is started when a discontact (DISC) is sent. It is stopped when an unnumbered acknowledgment (UA) or disconnect mode (DM) is received. If the number of sent DISCs reaches the retry limit, an outage is generated.

This timer is not configurable. The discontact time-out is 1000 milliseconds; the discontact retry limit is 3.

dataru[s+36..s+37] Negative poll time-out
dataru[s+38..s+39] Negative poll retry limit

This timer is used for primary SDLC only. At intervals specified by the negative poll time-out, a receive ready (RR) is transmitted. The negative poll retry limit is set at no limit; therefore, no outage is generated, no matter how many RRs are transmitted without acknowledgment being received.

The negative poll time-out is configurable in SNA Manager, in the advanced settings for an SDLC connection, where the time-out is called poll rate. Poll rate is set in polls per second (and translated internally into the negative poll time-out, timed in milliseconds).

The negative poll retry limit is not configurable. It is set at –1, meaning no limit.

dataru[s+40..s+41] T1 (no acknowledgment) time-out
dataru[s+42..s+43] N2 (no acknowledgment) retry limit

This timer is used for primary SDLC and is started when a poll/final bit is expected. If the time-out expires before a frame containing a poll/final bit is received, an RR is sent. When the number of sent RRs reaches the retry limit, an outage is generated.

This timer is configurable in SNA Manager, in the advanced settings for an SDLC connection, where it is called the poll time-out and poll retry limit.

dataru[s+44..s+45] Remote station busy time-out
dataru[s+46..s+47] Remote station busy retry limit

This timer is used for primary SDLC and is started when a receive not ready (RNR) is received. It is stopped when an RR is received. If the time-out expires the number of times specified by the retry limit, an outage is generated.

This timer is not configurable. The remote station busy time-out is 1000 milliseconds; the remote station busy retry limit is 30. Therefore, the time allowed before an outage is 30 seconds.

Link Timers (described for SDLC only)

dataru[s+48..s+49] Idle time-out
dataru[s+50..s+51] Idle retry limit

This timer is configurable in SNA Manager, in the advanced settings for an SDLC connection.

dataru[s+52..s+53] Nonproductive receive time-out
dataru[s+54..s+55] Nonproductive receive retry limit

This timer is used for secondary SDLC only and is started when any frame is received for this station. It is stopped when additional frames are received for this station. If the time-out expires the number of times specified by the retry limit, the link service causes a pop-up message, but does not generate an outage (because multidrop lines can be very slow).

This timer is not configurable. The nonproductive receive time-out is 1000 milliseconds (1 second); the nonproductive receive retry limit is 60. Therefore, the time allowed before a pop-up message is 60 seconds.

dataru[s+56..s+57] Write time-out
dataru[s+58..s+59] Write retry limit

This timer is started after an information frame has been transmitted to the hardware and stopped when the hardware acknowledges the frame. If the time-out expires the number of times specified by the retry limit, an outage is generated.

This timer is not configurable. The write time-out is 1000 milliseconds (one second); the write retry limit is 15. Therefore, the time allowed before an outage is 15 seconds.

dataru[s+60..s+61] Link connection time-out
dataru[s+62..s+63] Link connection retry limit

This timer is started when an open link for a leased line is received, and stopped when data set ready (DSR) is raised. If the time-out expires the number of times specified by the retry limit, an outage is generated.

This timer is not configurable. The link connection time-out is 1000 milliseconds (one second); the link connection retry limit is 300. Therefore, the time allowed before an outage is 300 seconds.

X.25 link data	Description
dataru[s+22]	Circuit type 0x00 PVC 0x01 SVC
dataru[s+23]	PVC alias, starting at 1 for lowest PVC channel number (reserved for SVC)
dataru[s+24..s+25]	PVC packet size (reserved for SVC)
dataru[s+26]	Default level 3 window size for PVC (reserved for SVC)
dataru[s+27]	Link role 0x00 Primary 0x01 Secondary 0x02 Negotiable

802.2 link data	Description
dataru[s+22]	Maximum receives without a transmit acknowledgment
dataru[s+23]	Maximum transmits without a receive acknowledgment
dataru[s+24]	Dynamic window increment value
dataru[s+25]	Remote SAP address
dataru[s+26]	Local SAP address (for incoming calls)
dataru[s+27]	Value for t1 timer multiplier

dataru[s+31]	Value for t2 timer multiplier
dataru[s+35]	Value for t3 timer multiplier
dataru[s+42]	Maximum retry count (N2 Value). Note that this 802.2 link data is required for the 802.2 command DLC.OPEN.STATION

Note that if the above fields for timer multipliers are set to zero, the SNALink should use appropriate defaults.

End of link data section	Description
dataru[s+68..s+69]	Length of link connection data (= a) (0x0000) None present
dataru[s+70..s+70+a]	Link connection data
dataru[s+70+b..s+71+b]	Where b is maximum of a and 20. Size of XID I-frame (= n) (0x0000) NULL XID
dataru[s+72+b]	XID

Note that if there are 20 or fewer bytes of link connection data, the XID length is at s+90 and the actual XID starts at s+92.

The link connection data can contain one of the following:

- MAC address of remote station
- X.25 address of remote station
- Dial-digits for manual or auto-dial modems

Optional Second Element (Only Used by X.25 SVC)

Element 2

Field	Type	Description
hdreptr–>elteptr–>elteptr	PTRBFELT	Pointer to next buffer element: NULL
hdreptr–>elteptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>elteptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>elteptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s]		Length of facilities data field (= c) inclusive of this length byte 0x01 no facilities data
dataru[s+1..s+c–1]		CHAR[c–1]Facilities data
dataru[s+c]		CHARLength of user data field (= d) inclusive of this length byte 0x01 no user data
dataru[s+c+1..s+c+d]		User data

Open(LINK) Response

Flow : DLC ------> NODE

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements: 1
msgtype	CHAR	Message type: OPENMSG (0x01)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
ophdr.openqual	CHAR	Open qualifier - RSPOK (0x02) - RSPERR (0x03)
ophdr.opentype	CHAR	Open type - LINK (0x10)
ophdr.opresid	INTEGER	Resource identifier
ophdr.operr1	INTEGER	Error code (see below)
ophdr.operr2	INTEGER	Reserved
hdreptr–>elteptr	PTRBFELT	Pointer to next buffer element: NULL
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+9]		Source name—same as destination name from **Open(LINK) Request**
dataru[s+10..s+19]		Destination name—name of local node; same as source name from **Open(LINK) Request**
dataru[s+22..s+23]		The maximum BTU size supported by SNALink, This size is 65,536 (largest number in an unsigned short) for channel connections and 32,768 for non-channel connections. Note that this limit does not guarantee that the SNA connection will actually use this value. The individual link service or the host can negotiate it downward.

The error codes (for an ERROR-RESPONSE) are defined as follows in SNA_CNST.H:

Symbolic constant	Value	Description
ERINIFAIL	0x01	Hardware initialization failed
ERINVXID	0x08	Invalid XID length
ERLINKOPN	0x09	Link already open
ERLLCERR	0x0A	LCC error; fatal hardware failure
ERBADINDX	0x0B	Invalid link index
ERBADOPN	0x0C	**Open(LINK)** has insufficient data
ERCONNTO	0x0D	Link connection time-out

ERNORES	0x0E	Maximum connection count reached –or– No more internal control blocks
EROPNPND	0x11	**Close(LINK)** arrived while **Open(LINK)** pending
ERDUPREQ	0x12	Duplicate request

Close(LINK)

Close(LINK) is used by the node to close the LINK LPI connection.

Close(LINK) Request

Flow : NODE ------> DLC

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: CLOSEMSG (0x02)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
clhdr.closqual	CHAR	Close qualifier: REQU (0x01)
clhdr.clstype	CHAR	Close type: LINK (0x10)

Note that the message consists of a buffer header only.

Close(LINK) Response

Flow : DLC ------> NODE

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: CLOSEMSG (0x02)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
clhdr.closqual	CHAR	Close qualifier - RSPOK (0x02) - RSPERR (0x03)
clhdr.clstype	CHAR	Close type: LINK (0x10)
clhdr.clserr1	INTEGER	Error code (see below)

The error codes (for an ERROR-RESPONSE) are defined as:

- 0x03—Link not open
- 0x04—Invalid link index

Note The message consists of a buffer header only.

Note The **Close(LINK)** message unconditionally shuts down the link.

Open(STATION)

Open(STATION) is used by the node to open the STATION LPI connection.

Open(STATION) Request

Flow : NODE ------> DLC

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements: 1
msgtype	CHAR	Message type: OPENMSG (0x01)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
ophdr.openqual	CHAR	Open qualifier: REQU (0x01)
ophdr.opentype	CHAR	Open type: STAT (0x11)
ophdr.opresid	INTEGER	Resource identifier
ophdr.icreditr	INTEGER	Initial credit for flow DLC –> NODE

Element

Field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to optional second buffer element (NULL if only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+9]		Source name—name of local node
dataru[s+10..s+19]		Destination name
dataru[s+20..s+21]		Link index as specified in **Open(LINK) Request**
dataru[s+22]		If NODE is primary, address of secondary station to initiate contact procedure with. 0x00 if NODE is secondary
dataru[s+23]		FID2 indicator 0x00 FID2 used
dataru[s+24]		Station type 0x00 Subarea 0x01 Peer
dataru[s+25..s+26]		Length of network name from received XID 0000 = No name
dataru[s+27..s+27+n]		Network name from received XID, in local character set, or if this is null, the name of the remote PU record in the COM.CFG file. This name can be fully qualified and has a maximum length of 17 characters.
dataru[s+44..s+89]		Link data—a copy of that supplied on the **Open(LINK) Request**

dataru[s+90..s+91]		The maximum BTU size to be used with this station. This size is 65,536 (largest number in an unsigned short) for channel connections and 32,768 for non-channel connections. Note that this limit does not guarantee that the SNA connection will actually use this value. The individual lin k service or the host can negotiate it downward.
dataru[s+m+1]		SAP used by remote station. The remote SAP information is only allowed for 802.2 connections, and may onl y be present if Signalling information is present It is used along with the Signalling Information to identify th e remote station.

Open(STATION) OK Response

Flow : DLC ------> NODE

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements: 1
msgtype	CHAR	Message type: OPENMSG (0x01)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
ophdr.openqual	CHAR	Open qualifier: RSPOK (0x02)
ophdr.opentype	CHAR	Open type: STAT (0x11)
ophdr.opresid	INTEGER	Resource identifier
ophdr.icreditr	INTEGER	Initial Credit for flow DLC –> NODE
ophdr.icredits	INTEGER	Initial Credit for flow NODE –> DLC

Element

Field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to next buffer element (NULL is only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+9]		Source name—same as destination name from **Open(STATION) Request**
dataru[s+10..s+19]		Destination name—name of local node; same as source name from **Open(STATION) Request**

Open(STATION) Error Response

Flow : DLC ------> NODE

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements: 1
msgtype	CHAR	Message type: OPENMSG (0x01)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
ophdr.openqual	CHAR	Open qualifier: RSPERR (0x03)
ophdr.opentype	CHAR	Open type: STAT (0x11)
ophdr.opresid	INTEGER	Resource identifier
ophdr.operr1	INTEGER	Error code
ophdr.operr2	INTEGER	Reserved

Element

Field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to next buffer element (NULL is only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array - 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd - 1
dataru[s..s+9]		Source name
dataru[s+10..s+19]		Destination name

The error codes are defined as follows:

Symbolic constant	Value	Description
ERLKNOTOPEN	0x03	Link not open
ERSTATOPEN	0x05	Station already open
ERNOCB	0x06	Station control blocks depleted
ERINVINDX	0x07	Invalid link index
ERMAXSTAT	0x08	Limit for number of stations per link reached
ERDIFADDR	0x09	Address different from that on **Request-Open-Station**
ERBADADDR	0x0A	Invalid DLC address

Close(STATION)

Close(STATION) is used by the node to close the STATION LPI connection.

Close(STATION) Request

Flow : NODE ------> DLC

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: CLOSEMSG (0x02)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
clhdr.closqual	CHAR	Close qualifier: REQU (0x01)
clhdr.clstype	CHAR	Close type: STAT (0x11)

Note that the message consists of a buffer header only.

Close(STATION) Response

Flow : DLC ------> NODE

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: CLOSEMSG (0x02)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
clhdr.clsequal	CHAR	Close qualifier - RSPOK (0x02) - RSPERR (0x03)
clhdr.clstype	CHAR	Close type: STAT (0x11)
clhdr.clserr1	INTEGER	Error code

The error codes (for an ERROR-RESPONSE) are defined as:

- 0x03—Station not open
- 0x04—Link not connected
- 0x05—Invalid station index
- 0x06—Duplicate request

Note The message consists of a buffer header only.

Note The Close(STATION) message unconditionally closes the station connection.

Request-Open-Station

Flow : DLC ------> NODE (link connection)

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element if present
numelts	CHAR	Number of buffer elements
msgtype	CHAR	Message type: DLCSTAT (0x11)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
dshdr.dstype	CHAR	Status type: QOPNSTN (0x16)
dshdr.dsqual	CHAR	Station address on XID or mode-set command. Set to 0x01 for 802.2
dshdr.dsmdset	CHAR	Rcv-Set-Mode flag 0x00 XID received 0x01 Mode set command received for example, SNRM for SDLC SABME for 802.2 QSM for X.25

Element field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to next buffer element (NULL is only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+n–1]		XID-starting at first byte of received XID information field. 0x00 NULL XID received (n = 1) and signaling information present.

Optional Signaling Information

Field	Description
dataru[s+n]	Length of data, including this bytes
dataru[s+n+1]	Type of data (not used at present)
dataru[s+n+2..s+m]	Address or other identifier data. For example, MAC address of remote station X.25 address of remote station

Note the following:

- The signaling information is used by the node to identify the remote station on 802.2 and X.25 links.
- If a NULL XID is received and no signaling information is required, the element can be omitted.
- If a NULL XID is received and signaling information is required, an 0x00 byte should be put in the element followed by the signaling information.
- If the Rcv-Set-Mode flag is set to 0x01, the element can be omitted.

Station-Contacted

Flow : DLC ------> NODE (station connection)

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: DLCSTAT (0x11)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
dshdr.dstype	CHAR	Status type: STNCTCTD (0x17)

Note that this message contains a buffer header only.

Outage

Flow : DLC ------> NODE (link or station connection)

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: DLCSTAT (0x11)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
dshdr.dstype	CHAR	Status type: OUTAGE (0x18)
dshdr.dsqual	CHAR	Outage qualifier
dshdr.dsoutsq	CHAR	Outage subqualifier (optional)

Note the following:

- This message contains a buffer header only.
- Outage qualifier codes are given in Outages.

Status-Resource

Flow : DLC <------> NODE (station connection)

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element: NULL
numelts	CHAR	Number of buffer elements: 0
msgtype	CHAR	Message type: DLCSTAT (0x11)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
dshdr.dstype	CHAR	Status type: RESOURCE (0x04)
dshdr.dlccred	INTEGER	DLC credit

Note the following:

- This message contains a buffer header only.
- The **dlccred** field indicates that the message sender can receive a further **dlccred DLC-Data** messages.

Send-XID

Flow : NODE ------> DLC (link connection)

Header

Field	Type	Description
nxtqptr	PTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PTRBFELT	Pointer to first buffer element
numelts	CHAR	Number of buffer elements
msgtype	CHAR	Message type: DLCSTAT (0x11)
srcl	CHAR	Source locality
srcp	CHAR	Source partner
srci	INTEGER	Source index
destl	CHAR	Destination locality
destp	CHAR	Destination partner
desti	INTEGER	Destination index
dshdr.dstype	CHAR	Status type: SENDXID (0x1A)
dshdr.dsqual	CHAR	Station address on XID

Element

Field	Type	Description
hdreptr–>elteptr	PTRBFELT	Pointer to next buffer element: (NULL is only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+n–1]		XID information frame.

Note that the **dshdr.dsqual** field is valid only for primary multipoint connections where station is specified on a multidrop line that the XID should be sent to. In all other cases, it is set to 0xFF.

In situations where the link protocol requires the address field on the XID to be set to a value other than 0xFF (for example, to specify that the XID is a response), it is the responsibility of the link service to set this byte appropriately.

The **Send-XID** message can contain zero elements (**numelts** = 0) or a single, empty element (**hdreptr–>startd** < **hdreptr–>endd**). In these cases, the link service is expected to transmit a NULL XID.

DLC-Data

Flow : DLC <------> NODE

Header

Field	Type	Description
nxtqptr	PPTRBFHDR	Pointer to next buffer header in a queue
hdreptr	PPTRBFELT	Pointer to first buffer element
numelts	CCHAR	Number of buffer elements
msgtype	CCHAR	Message type: DLCDATA (0x10)
srcl	CCHAR	Source locality
srcp	CCHAR	Source partner
srci	INTEGER	Source index
destl	CCHAR	Destination locality
destp	CCHAR	Destination partner
desti	INTEGER	Destination index
ddhdr.ddth01	CCHAR[6]	Transmission header

Element

Field	Type	Description
hdreptr–>elteptr	PPTRBFELT	Pointer to next buffer element: (NULL is only one element)
hdreptr–>startd	INTEGER	Index to start of data in this buffer element's data array: 1
hdreptr–>endd	INTEGER	Index to last byte of data in this buffer element's data array
hdreptr–>dataru	CHAR[SNANBEDA]	Defined as follows, where s = startd–1
dataru[s..s+n–1]		SNA request header (RH) if present, and request unit (RU)

Configuration Entry Points

The following topics describe the entry points used by the SNALink to obtain configuration information.

SNAGetConfigValue

The SNALink calls the **SNAGetConfigValue** function to obtain the value of a specific configuration parameter.

```
USHORT SNAGetConfigValue(
UCHAR *entryName,
VOID *pBuffer,
ULONG bufferLen,
UCHAR parmType,
ULONG *pRetLength
);
```

Parameters

entryName
The name of the configuration parameter required.
pBuffer
A pointer to a buffer (if parameter is a string), or a pointer to a LONGINT (if parameter is an integer).
bufferLen
The length of the buffer. Only required if the parameter is TYPESTRING.
parmType
TYPESTRING if parameter is a string.

TYPELONG if parameter is an integer.

pRetLength
Number of bytes returned if parameter is TYPESTRING, or number of bytes available if the buffer was too short.

Return Values

NO_ERROR
OK
ERBADCFG
Error reading configuration file.
ERNOTFND
Entry not found in configuration record.
ERTOOLONG
Data available exceeded the size of the buffer.
ERBADTYPE
A bad type was specified for the *parmType* parameter.

Remarks

It is strongly recommended that the SNALink read all required configuration parameters at initialization time (when SNALinkInitialize is called by the Base).

SNAGetSystemInfo

The SNALink calls the **SNAGetSystemInfo** function to obtain information about SNAServer and the network operating system.

```
INTEGER SNAGetSystemInfo(
struct cs_info *pCSInfo
);
```

Parameters

pCSInfo
Pointer to buffer supplied by application, containing a data structure in which system information is returned. The application must set the **length** field in this data structure (see Remarks); the other fields should be set to nulls or blanks.

```
struct cs_info {
    unsigned short  length;
    unsigned char   major_ver;
    unsigned char   minor_ver;
    unsigned char   config_share[80];
    unsigned short  nos;
};
```

Members

length
Length of the data structure supplied by the application.

major_ver
Major version number:

1 for Comm Server 1.1

2 for SNA Server 2.0/2.1

3 for SNA Server 3.0

4 for SNA Server 4.0

minor_ver
Minor version number (decimal):

10 for Comm Server 1.1

00 for SNA Server 2.0

20 for SNA Server 2.1

00 for SNA Server 3.0

00 for SNA Server 4.0

config_share[80]
The name of the share point of the current configuration file (\\server\share\, for example). This path name must be a null-terminated string.

nos
Transport protocol in use:

bit 0: LAN Manager/LAN Server (named pipes)

bit 1: NetWare (IPX/SPX)

bit 2: AppleTalk

bit 3: Banyan VINES (VINES IP)

bit 4: TCP/IP

Return Values

NO_ERROR

OK
ERNOCFGSVR
No configuration file server available.
ERMOREDATA
Supplied buffer was too small.

Remarks

The application must set the **length** parameter to the length of the **cs_info** structure (86 bytes in the current version). Any other value will be rejected. This parameter is used to ensure compatibility with future versions; an application supplying this length will always obtain the information shown here, but in future versions it may be possible to specify larger values and obtain further information.

On successful return, the **cs_info** data structure contains the version number of SNA Server (SNA Server 2.x, SNA Server 3.0, or Comm Server 1.x, for older versions), the path to the current configuration file, and the network operating system over which SNA Server is running.

If there is no configuration file server available, only the version number fields are valid; the other fields should not be checked.

Setup Functions

This section provides a reference for the functions used with the newer integrated link service DLL architecture or with the older INF-based setup design.

Integrated Link Service Configuration Functions

This section provides a reference for exported DLL entry points and utility functions used when building an integrated link service configuration DLL.

Functions Exported from a Link Service Configuration DLL

This section provides a reference for functions that must be exported from an integrated link service configuration DLL.

CommandLineAdd

The **CommandLineAdd** function is used to add a new link service using a command-line interface. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI CommandLineAdd(
  LPSTR szCommandLine,
  LPSTR *szConfigInfo,
  LPDWORD dConfigInfoSize
);
```

Parameters

szCommandLine
This supplied parameter specifies the command line containing information on the computer and link service to be configured.
szConfigInfo
This supplied and returned parameter points to a configuration buffer that is used to configure the link service.
dConfigInfoSize
This supplied parameter specifies the size of the sz*ConfigInfo* configuration buffer .

Return Values

TRUE
The function executed successfully.
FALSE
One or more of the parameters passed to this function are invalid or the function failed.

ConfigureLinkService

The **ConfigureLinkService** function is used to add or modify a link service. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI ConfigureLinkService(
  LPSTR szComputerName,
  LPSTR szLinkServiceTitle
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer that is to be configured.
szLinkServiceTitle
This supplied parameter specifies the title of the link service that is to be configured.

Return Values

TRUE
The function executed successfully and network bindings need to be recalculated.
FALSE
One or more of the parameters passed to this function are invalid or network bindings do not need to be recalculated.

ConfigureLinkServiceEx

The **ConfigureLinkServiceEx** function is used to add or modify a link service. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI ConfigureLinkServiceEx(
  LPSTR szComputerName,
  LPSTR szLinkServiceTitle,
  LPSTR* pvConfigInfo,
  LPDWORD dConfigInfoSize
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer that is to be configured.
szLinkServiceTitle
This supplied parameter specifies the title of the link service that is to be configured.
pvConfigInfo
This supplied and returned parameter points to a configuration buffer that is used to configure the link service.
dConfigInfoSize
This supplied parameter specifies the size of the *pvConfigInfo* configuration buffer.

Return Values

TRUE
The function executed successfully and network bindings need to be recalculated.
FALSE
One or more of the parameters passed to this function are invalid or network bindings do not need to be recalculated.

DisplayHelpInfo

The **DisplayHelpInfo** function is used to generate help information used by the command-line interface to a link service DLL. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI DisplayHelpInfo(
  LPSTR * szHelpInfoBuffer
);
```

Parameters

szHelpInfoBuffer
This supplied and returned parameter points to a buffer that on successful return contains help information that can be used to configure the link service.

Return Values

TRUE
The function executed successfully.
FALSE
The parameter passed to this function is invalid or the function failed.

RemoveAllLinkServices

The **RemoveAllLinkServices** function is used to remove all link services from a machine. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI RemoveAllLinkServices(
  LPSTR szComputerName
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer that is to have all link services removed.

Return Values

TRUE
The function executed successfully and network bindings need to be recalculated.

FALSE
The parameter passed to this function is invalid or network bindings do not need to be recalculated.

RemoveLinkService

The **RemoveLinkService** function is used to remove a link service. This function must be exported from a link service configuration DLL supplied with each link service.

```
__declspec(dllexport) BOOL WINAPI RemoveLinkService(
  LPSTR szComputerName,
  LPSTR szLinkServiceTitle
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer that is to have the link service removed.
szLinkServiceTitle
This supplied parameter specifies the title of the link service that is to be removed.

Return Values

TRUE
The function executed successfully and network bindings need to be recalculated.
FALSE
One or more of the parameters passed to this function are invalid or network bindings do not need to be recalculated.

Utility Functions Used by a Link Service Configuration DLL

This section provides a reference for utility functions used by an integrated link service configuration DLL.

AddPerfmonCounters

The **AddPerfmonCounters** function is used to add perfmon counters to a link service. This utility function is used to construct an integrated link service configuration DLL.

```
void AddPerfmonCounters(
  LPSTR pszComputerName,
  LPSTR pszService
);
```

Parameters

pszComputerName
This supplied parameter specifies the name of the computer that is to have perfmon counters added.
pszService
This supplied parameter specifies the name of the link service that is to have perfmon counters added.

Return Values

None.

Remarks

SNA RPC Service must be running or an error MessageBox will indicate a failure.

bCreateService

The **bCreateService** function is used to create a service on a computer for a link service. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL bCreateService(
  LPSTR szComputerName,
  LPSTR szServiceName,
  LPSTR szServicePath,
  LPSTR szServiceDependencies,
  DWORD dServiceType,
  DWORD dServiceLoadType,
  LPSTR szDomainName,
  LPSTR szUserid,
  LPSTR szPassword
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer to create the service on.

szServiceName
This supplied parameter specifies the name of the link service that is to be created. This parameter is passed unchanged to the Windows 2000/NT **CreateService** function.

szServicePath
This supplied parameter specifies the binary path to the link service that is to be created. This parameter is passed unchanged to the Windows 2000/NT **CreateService** function.

szServiceDependencies
This supplied parameter specifies the service dependencies of the link service that is to be created. This parameter is passed unchanged to the Windows 2000/NT **CreateService** function.

dServiceType
This supplied parameter specifies the type of service that is to be created. This parameter is passed unchanged to the Windows 2000/NT **CreateService** function.

dServiceLoadType
This supplied parameter specifies the load type of service that is to be created. This parameter is passed unchanged to the Windows 2000/NT **CreateService** function.

szDomainName
This supplied parameter specifies the domain name for the service to run in.

szUserid
This supplied parameter specifies the user identifier for the service to run in.

szPassword
This supplied parameter specifies the password for the domain account.

Return Values

TRUE
The function executed successfully and the service was created.

FALSE
One or more of the parameters passed to this function are invalid or the function failed.

Remarks

If the *szUserid* parameter is not supplied, then the *szDomainName* parameter is used to construct the Account parameter passed to the Windows 2000/NT **CreateService** function.

bDeleteService

The **bDeleteService** function is used to delete a service on a computer for a link service. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL bDeleteService(
  LPSTR szComputerName,
  LPSTR szServiceName
);
```

Parameters

szComputerName
This supplied parameter specifies the name of the computer to delete the service on.

szServiceName
This supplied parameter specifies the name of the service that is to be deleted. This parameter is passed unchanged to the Windows 2000/NT **OpenService** function.

Return Values

TRUE
The function executed successfully and the service was deleted.

FALSE
One or more of the parameters passed to this function are invalid or the function failed.

bStopService

The **bStopService** function is used to stop a service running on a computer for a link service. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL bStopService(
  LPSTR szServiceName,
  LPSTR szComputerName
);
```

Parameters

szServiceName
This supplied parameter specifies the name of the service that is to be stopped. This parameter is passed unchanged to the Windows 2000/NT **OpenService** function.

szComputerName
This supplied parameter specifies the name of the computer to stop the service on.

Return Values

TRUE
The function executed successfully and the service was stopped.

FALSE
One or more of the parameters passed to this function are invalid or the function failed.

CheckForExistingLinkService

The **CheckForExistingLinkService** function is used to check to see if a link service of this type exists with this title. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL bCreateService(
  HKEY *hGlobalKey,
  LPSTR szLinkRegistryRoot,
  LPSTR szTitle
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry to search.
szLinkRegistryRoot
This supplied parameter specifies the registry root for this type of link service to search for.
szTitle
This supplied parameter specifies the title of the link service to search for.

Return Values

TRUE
The link service was located.
FALSE
One or more of the parameters passed to this function are invalid or the link service was not located.

ConvertHexStringToDWORD

The **ConvertHexStringToDWORD** function is used to convert a hexadecimal string to a DWORD value. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL ConvertHexStringToDWORD(
  LPSTR szHexString,
  LPDWORD dHexValue
);
```

Parameters

szHexString
This supplied parameter specifies the hexadecimal string to be converted.
dHexValue
This supplied and returned parameter contains the DWORD value of the hexadecimal string if the function was successful.

Return Values

TRUE
The function executed successfully and the hexadecimal string was converted.
FALSE
One or more of the parameters passed to this function are invalid or the function failed.

Remarks

This function scans until a nonhexadecimal character is encountered. The hexadecimal formats recognized are xnnnn, 0xnnnn, or nnnn.

ExtractNextParameter

The **ExtractNextParameter** function is used to get the next parameter from a buffer. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL ExtractNextParameter(
  LPSTR szSourceBuffer,
  LPSTR szParameter,
  LPDWORD dStartIndex
);
```

Parameters

szSourceBuffer
This supplied parameter specifies the source buffer.
szParameter
This supplied and returned parameter specifies the return buffer for parameters.
dStartIndex
This supplied parameter contains the DWORD index to begin parameter scan.

Return Values

TRUE
The function executed successfully and the next parameter was located and returned in the *szParameter* argument.
FALSE
The function failed.

Remarks

Parameters are delimited by a space character. Quotes can be used to include spaces in a parameter.

fAddRegistryEntry

The **fAddRegistryEntry** function is used to add a new registry value to the registry. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fAddRegistryEntry(
  HKEY *hGlobalKey,
  char *szRegistryValue,
  char *szRegistryData,
  DWORD dType,
  DWORD dSize
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry to modify.
szRegistryValue
This supplied parameter specifies the registry value name to add.
szRegistryData
This supplied parameter specifies the registry value data to add.
dType
This supplied parameter specifies the registry value type. This parameter is supplied unchanged to the Win32 ® **RegSetValueEx** function.
dSize
This supplied parameter specifies the size of the registry value data. This parameter is supplied unchanged to the Win32 **RegSetValueEx** function.

Return Values

TRUE
The function executed successfully and the registry entry was added.
FALSE
The function failed and the registry entyr was not added.

fCanWeAdministerRemoteBox

The **fCanWeAdministerRemoteBox** function is used to determine if the caller has administrative privileges on the remote computer. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fCanWeAdministerRemoteBox(
  HKEY *hGlobalKey
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle to the remote computer's registry.

Return Values

TRUE
The caller has administrative privileges on the remote computer.
FALSE
The caller lacks administrative privileges.

fConnectRegistry

The **fConnectRegistry** function is used to connect to a remote computer's registry and return a handle to the remote registry. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fConnectRegistry(
  HKEY *hGlobalKey,
  LPSTR *szComputerName
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry to connect to.

szComputerName
This supplied parameter specifies the computer name to connect to.

Return Values

TRUE
The function executed successfully and the function was able to connect to the registry.

FALSE
The function failed.

fDisConnectRegistry

The **fDisConnectRegistry** function is used to disconnect from a remote computer's registry. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fDisConnectRegistry(
  HKEY *  hGlobalKey
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry to disconnect from.

Return Values

TRUE
The function executed successfully and the function was able to disconnect from the registry.
FALSE
The function failed.

fFindAndReplaceString

The **fFindAndReplaceString** function is used to find and replace a substring within a string. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fFindAndReplaceString(
  LPSTR szStringBuffer,
  LPSTR szSearchString,
  LPSTR szReplaceString
);
```

Parameters

szStringBuffer
This supplied parameter specifies the string buffer to search.
szSearchString
This supplied parameter specifies the string to search for.
szReplaceString
This supplied parameter specifies the replacement string.

Return Values

TRUE
The string was found.
FALSE
The string was not found.

fFindString

The **fFindString** function is used to determine if a string exists within a string buffer. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fFindString(
  LPSTR szStringBuffer,
  LPSTR szSearchString
);
```

Parameters

szStringBuffer
This supplied parameter specifies the string buffer to search.
szSearchString
This supplied parameter specifies the string to search for.

Return Values

TRUE
The string was found.
FALSE
The string was not found.

fFindStringInMultiSZ

The **fFindStringInMultiSZ** function is used to determine if string exists in a REG_MULTI_SZ string list and return entire string. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fFindString(
  LPSTR szStringBuffer,
  LPSTR szSearchString,
  LPSTR szFoundString
);
```

Parameters

szStringBuffer
This supplied parameter specifies the string buffer to search.
szSearchString
This supplied parameter specifies the string to search for.
szFoundString
This supplied and returned parameter specifies the full string containing string if successful.

Return Values

TRUE
The string was found and returned.
FALSE
The string was not found.

fQueryRegistryValue

The **fQueryRegistryValue** function is used to query a value from the registry.This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fQueryRegistryValue(
  HKEY *  hGlobalKey,
  LPSTR szRegistryKey,
  LPSTR szRegistryValue,
  LPSTR szRegistryData,
  LPDWORD dSize
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry.
szRegistryKey
This supplied parameter specifies the registry key.
szRegistryValue
This supplied parameter specifies the registry value name.
szRegistryData
This supplied parameter specifies the registry value data.
dSize
This supplied parameter specifies the size of the registry data.

Return Values

TRUE
The function executed successfully and the function was able to connect to the registry.
FALSE
The function failed.

fRegistryKeyExists

The **fRegistryKeyExists** function is used to determine if a registry key already exists in the registry.This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fRegistryKeyExists (
  HKEY *  hGlobalKey,
  LPSTR szRegistryKey
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry.
szRegistryKey
This supplied parameter specifies the registry key.

Return Values

TRUE
The registry key exists.
FALSE
The function failed or the registry key doen't exist.

fRemoveRegistryEntry

The **fRemoveRegistryEntry** function is used to remove a registry key from the registry.This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fRemoveRegistryEntry (
  HKEY *  hGlobalKey,
  char * szRegistryKey
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry.
szRegistryKey
This supplied parameter specifies the registry key.

Return Values

TRUE
The function was successful and the registry key was removed.
FALSE
The function failed or the registry key could not be removed.

fRemoveRegistryValue

The **fRemoveRegistryValue** function is used to remove a registry value from the registry.This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fRemoveRegistryValue (
  HKEY *  hGlobalKey,
  char * szRegistryKey,
  char * szRegistryValue
);
```

Parameters

hGlobalKey
This supplied parameter specifies the handle of the registry.
szRegistryKey
This supplied parameter specifies the registry key.
szRegistryValue
This supplied parameter specifies the registry value to remove.

Return Values

TRUE
The function was successful and the registry value was removed.
FALSE
The function failed or the registry value could not be removed.

fStringCompare

The **fStringCompare** function is used to determine if string exists in another string. This utility function is used to construct an integrated link service configuration DLL.

```
BOOL fStringCompare (
  LPSTR lpszString1,
  LPSTR lpszString2
);
```

Parameters

lpszString1
This supplied parameter specifies the string to search for.
lpszString2
This supplied parameter specifies the string to compare.

Return Values

TRUE
The string was found.
FALSE
The string was not found.

LoadStringResource

The **LoadStringResource** function is used to load a string from a string resource. This utility function is used to construct an integrated link service configuration DLL.

```
void LoadStringResource (
  DWORD dStringResource,
  LPSTR pszString
);
```

Parameters

dStringResource
This supplied parameter specifies the resource ID of the string resource.

pszString
This supplied and returned parameter specifies the buffer to place the string in.

Return Values

None

ParseNextField

The **ParseNextField** function is used to parse and return the next field from string. This utility function is used to construct an integrated link service configuration DLL.

```
void ParseNextField(
  LPSTR szParseString,
  LPSTR szField,
  char scDelimiter,
  LPDWORD dStartIndex
);
```

Parameters

szParseString
This supplied parameter specifies the string to parse.
szField
This supplied and returned specifies the return buffer for the next field.
cDelimiter
This supplied parameter specifies the delimiter character for the end of a field.
dStartIndex
This supplied parameter specifies a pointer to the index in bytes from beginning of the *szParseString* to start the search from.

Return Values

TRUE
The next field was found.
FALSE
The next field was not found.

Note The **'^'** character can be used as an escape character to allow the delimiter to be used.

RemovePerfmonCounters

The **RemovePerfmonCounters** function is used to remove counters from a link service. This utility function is used to construct an integrated link service configuration DLL.

```
void RemovePerfmonCounters(
  LPSTR pszComputerName,
  LPSTR pszService
);
```

Parameters

pszComputerName
This supplied parameter specifies the name of the computer that is to have perfmon counters removed.
pszService
This supplied parameter specifies the name of the link service that is to have perfmon counters removed.

Return Values

None.

Remarks

SNA RPC Service must be running or an error MessageBox will indicate a failure.

INF-Based Setup Functions

This section provides a reference for some of the useful entry points in the .INF file that contains utility functions. The file name is in the variable *UtilityInf*; usually set to SNAUTILS.INF.

CreateSNARegEntry

The **CreateSNARegEntry** function creates the necessary entries for an instance in the **SOFTWARE\Microsoft** registry tree. If the product is not already in the registry, it creates an entry for the product. It then creates an entry for the particular instance of the product and for the **NetRules** key under that entry. This function leaves open handles to all the important subkeys for further use.

Parameters

Argument 0
Name of the top-level registry node to use. This should be a full registry path. For most scenarios, this is the value held in the *ProductRegBase* variable (**SOFTWARE\Microsoft**).
Argument 1
Name of the product. This is the name of the key that will be created for this product.
Argument 2
Instance index. This is the index of this particular instance of this product.

Return Values

Return 0
Status of the operation:

- STATUS_SUCCESSFUL: Operation succeeded.
- STATUS_FAILED: Operation failed.

Return 1
Handle to the top-level registry node.
Return 2
Handle to the product's registry key under the top-level node.
Return 3
Handle to the instance entry under the product key.
Return 4
Handle to the NetRules entry under the instance key.

CreateSNAService

The **CreateSNAService** function creates the necessary entries for an instance in the **Services** registry tree. This function adds particular values that are necessary for the service as well as all the subkeys under the service key, including **Parameters** and **ExtraParameters**.

Parameters

Argument 0
Name of the service to be created.
Argument 1
Type of SNA Service (*SNAServiceType* variable).
Argument 2
Image path of this component (*ImagePath* variable).
Argument 3
Dependency list (*ProductDepends* variable).
Argument 4
Parameter list (*ProductParams* variable).
Argument 5
Extra parameter list (*ProductExtraParams* variable).
Argument 6
Event Log message file.
Argument 7
Event types supported (usually 0x07).

Return Values

Return 0
Status of the operation:

- STATUS_SUCCESSFUL: Operation succeeded.
- STATUS_FAILED: Operation failed.

Return 1
Handle to the service key.
Return 2
Handle to the **Parameters** key under the service key.
Return 3
Handle to the **ExtraParameters** key under the **Parameters** key.

DeleteSNAService

The **DeleteSNAService** function deletes a particular service using the Service Control Manager. All the keys for the service are deleted as well.

Parameters

Argument 0
Name of the service to be deleted.

Return Values

Return 0
Status of the operation.

EnterServiceName

The **EnterServiceName** function presents the user with an algorithmically determined service name for a component and allows the user to change it before returning the final value. This function ensures that the new service name is unique in the Service Control Architecture before accepting it.

Parameters

Argument 0
Title of the product the user should be queried about.
Argument 1
Default service name prefix.
Argument 2
Index for this instance of the product. The algorithm uses this index to determine the default name.

Return Values

Return 0
Status of the operation:

- STATUS_SUCCESSFUL: Operation succeeded.
- STATUS_NOSUCHLANGUAGE: The language specified is not supported.
- STATUS_USERCANCEL: User pressed the **Cancel** button.

Return 1
Service name that the user entered.

FindNextAvailableIndex

The **FindNextAvailableIndex** function is used to determine the index a new instance should receive. For example, if the current indexes in use are { 01, 02, 04 }, this function would return 03 as its return value.

Parameters

Argument 0
A list of the indexes currently in use. This list can be obtained by using the FindSNAProductServices function.

Return Values

Return 0
Status of the operation:

- STATUS_SUCCESSFUL: Operation succeeded.
- STATUS_FAILED: Operation failed.

Return 1
First available index for the list.

FindSNAProductServices

The **FindSNAProductServices** function enumerates all instances of a product. It returns lists of information for the instances that can be used in the script. This function can also be used to determine whether or not a product exists in the registry by analyzing the return status.

Parameters

Argument 0
Name of top-level registry node to use.
Argument 1
Name of the product.

Return Values

Return 0
Status of the operation:

- STATUS_SUCCESSFUL: Operation succeeded.
- STATUS_NOSUCHPRODUCT: The product does not exist in the registry.
- STATUS_FAILED: Operation failed.

Return 1
List of indexes for the instances of this product.
Return 2
List of service names for the instances of this product.
Return 3
List of titles for the instances of this product.
Return 4
List of descriptions for the instances of this product.

FindSNARegEntry

The **FindSNARegEntry** function is written as a parallel to CreateSNARegEntry. When called, it attempts to open all of the necessary registry keys and return open handles to them.

Parameters

Argument 0
Name of the top-level registry node to use.
Argument 1
Name of the product.
Argument 2
Instance index.

Return Values

Return 0
Status of the operation.
Return 1
Handle to the top-level registry node.
Return 2
Handle to the product's registry key under the top-level node.
Return 3
Handle to the instance entry under the product key.
Return 4
Handle to the NetRules entry under the instance key.

FindSNAService

The **FindSNAService** function is written as a parallel to CreateSNAService. It is written to provide an easy way to access the keys for a particular service.

Parameters

Argument 0
Name of the service to look for.

Return Values

Return 0
Status of the operation.
Return 1
Handle to the service key.
Return 2
Handle to the **Parameters** key under the service key.
Return 3
Handle to the **ExtraParameters** key under the **Parameters** key.

GrepUniqueServiceInfo

The **GrepUniqueServiceInfo** function is used to determine information about a particular instance when only one of the four elements (index, name, title, or description) is available. Because there is no way to determine the position of an element in a list, it is hard to figure out the respective name, title, or description of an instance given only the index. This function searches the list and returns the rest of the information about the instance.

Parameters

Argument 0
Type of information to search:

- INDEX: Search the list of indexes.
- NAME: Search the list of service names.
- TITLE: Search the list of titles.
- DESC: Search the list of descriptions.

Argument 1
Keyword to search for in the list.
Argument 2
List of indexes for the instances of this product.
Argument 3
List of service names for the instances of this product.
Argument 4
List of titles for the instances of this product.
Argument 5
List of descriptions for the instances of this product.

Return Values

Return 0
Status of the operation.
Return 1
Index for this instance of the product.
Return 2
Service name for this instance of the product.
Return 3
Title for this instance of the product.
Return 4
Description for this instance of the product.

SetupMessage

The **SetupMessage** function displays a dialog box with user-defined text plus **OK** and **Cancel** buttons. It is useful for debugging.

Parameters

Argument 0
Language to use (*STF_LANGUAGE*).
Argument 1
Which icon to display in the dialog box: STATUS, WARNING, NONFATAL, and so on.
Argument 2
The text to be displayed. Can contain linefeeds using the LF symbol.

Return Values

Return 0
Status of the operation.

IOCTL Commands

This section provides reference information about the IOCTL functions.

Function 0x41: Set Event/Semaphore Handle

This function supplies the driver with the handle of an event that can be used for signaling the SNALink software.

Parameters

IRP.UserEvent
This parameter is taken from the IOCTL call, and is an event handle. The handle must refer to an Event/Semaphore owned by the SNALink process. The driver sets the event to indicate the completion of a transmission or the availability of received data or status. Although not required by the driver, the event passed here by the SNALink is normally the global Base event.

Return Values

If the supplied parameter is NULL, the function returns a status of STATUS_INVALID_PARAMETER, with additional information of INFO_SET_EVENT_NO_EVENT. For any other illegal parameter, an exception is raised.

Remarks

This function should be called only once, immediately after the **OPEN** is issued. The event is set when:

- IFrames are transmitted from the driver buffers.
- IFrames are received into the driver buffers.
- IStatus information is updated in the Interface Record (see function 0x64).

Function 0x42: Set Link Characteristics

This function sets the link protocol parameters required by the driver.

Parameters

Frame Size (packet format WORD)
Indicates to the driver the minimum amount of contiguous receiver buffering that must be available for any individual frame.

Data Rate (packet format DWORD)
Line speed in bits per second. If *Link Options* bit 3 is not set, this field is ignored.

Station Address 1 (packet format BYTE)
Station Address 2 (packet format BYTE)
Station addresses that the user wishes to receive on if selective reception is to be used (typically for multidropped secondaries). Only frames of data beginning with either of these values will be passed to the user — others are ignored or discarded.

If the SNALink wishes to listen on only one station address, *Station Address 2* should be set to zero.

A value of zero in both fields indicates that all error-free received frames are to be passed to the SNALink, regardless of the contents of their first address byte.

Link Options (packet format BYTE)
Link Options is a bitmap. The default is all values set to zero. The bits are used as shown in the following table. Note that not all of these options are supported by the standard Host Integration Server 2000 synchronous card drivers.

Bit 7	1 = Four wire (constant RTS/CTS). 0 = Two wire (switched RTS/CTS).
Bit 6	1 = NRZI encoding. 0 = NRZ encoding.
Bit 5	1 = HDLC level 1 conventions. 0 = SDLC level 1 conventions.
Bit 4	1 = Full-duplex (simultaneous 2-way) data. 0 = Half-duplex (alternating 1-way) data.
Bit 3	1 = Generate internal clocking. 0 = Take external clocking.
Bit 2	1 = Use DMA if available. 0 = Do not use DMA on this link.
Bit 1	1 = Reset all statistics to zero. 0 = Leave accumulated statistics as is.
Bit 0	Reserved.

Reserved (packet format BYTE).

Packet Formats

WORD	Frame Size
DWORD	Data rate
BYTE	Station address 1
BYTE	Station address 2
BYTE	Link Options
BYTE	Reserved

Frame Size indicates to the driver the minimum amount of contiguous receiver buffering that must be available for any individual frame.

Data Rate is the line speed in bits per second. If Link Options bit 3 is not set, this field is ignored.

Station Address 1 and 2 are the station addresses that the user wishes to receive on if "selective" reception is to be used (typically for multi-dropped secondaries). Only frames of data commencing with either of these values will be passed up to the user — others are ignored or discarded.

If the SNALink wishes to listen on only one station address, station address 2 should be set to zero.

A value of zero in both fields indicates that all error-free received frames are to be passed to the SNALink, regardless of the contents of their first address byte.

Link Options is a bitmap. The default is all values set to zero. The bits are used as follows:

Bit 7 set to	1 = Four Wire (constant RTS/CTS) 0 = Two Wire (switched RTS/CTS)
Bit 6 set to	1 = NRZI encoding 0 = NRZ encoding
Bit 5 set to	1 = HDLC level 1 conventions 0 = SDLC level 1 conventions
Bit 4 set to	1 = Full-duplex (simultaneous 2-way) data 0 = Half-duplex (alternating 1-way) data
Bit 3 set to	1 = Generate internal clocking 0 = Take external clocking
Bit 2 set to	1 = Use DMA if available 0 = Don't use DMA on this link
Bit 1 set to	1 = Reset all statistics to zero 0 = Leave accumulated statistics as is
Bit 0 reserved.	

Note Not all of the above options are supported by the standard Host Integration Server 2000 synchronous card drivers.›

Return Values

IoStatus.Status	IoStatus.Information	Description
STATUS_INVALID_PARAMETER	IO_ERR_LINKCHARBUF_ WRONG_SIZE	
STATUS_INVALID_PARAMETER	IO_ERR_FRAME_BUF_ TOO_SMALL	Buffer must be at least 268 bytes.
STATUS_INVALID_PARAMETER	IO_ERR_FRAME_BUF_ TOO_BIG	Buffer maximum size is 2048 bytes.
STATUS_INVALID_PARAMETER	IO_ERR_NO_CLOCKS	No internal clocking available
STATUS_DATA_ERROR	IO_ERR_HARDWARE_ 8273CMD_TIMEOUT	
STATUS_SUCCESS	IO_ERR_NO_DMA_FDX	DMA requested, but can't be used

Remarks

The driver should always start the receiver after processing this request. If either the transmitter or receiver is active when this request is issued, the driver stops the current frame before resetting the link characteristics, then restarts the previous operation.

Link Service DLLs that support the synchronous dumb card interface use the following registry entries to the control this feature.

SYSTEM\CurrentControlSet\Services\<linkService>\Parameters

where <linkService> is the name of the link service.

Under this node, the following entries and values must be entered or modified:

A node called **ExtraParameters** must be created or modified with the following registry entries and values:

InternalClock
The value of this entry should be defined and set to a REG_DWORD of any value to enable the internal clock.
InternalClockRate
The value of this entry should be set to a REG_DWORD equal to the number of bits per second.

Function 0x43: Set V24 Output Status

This function allows the SNALink software to alter the modem output status on the adapter V.24 interface. There is no parameter or data packet on this request. The relevant V.24 settings are put into the driver interface record (see function 0x61) by the SNALink prior to calling the driver.

Return Values

IoStatus.Status	IoStatus.Information
STATUS_DATA_ERROR	IO_ERR_HARDWARE_8273CMD_TIMEOUT

Function 0x44: Transmit Frame

The SNALink calls this function to transfer a frame of data to the driver.

Parameters

IRP.CurrentStackLocation.OutputBufferLength
Frame length — the size of the frame pointed to by the data buffer pointer. The frame includes the control, address, and information field (if present), but no allowance should be made for flags or CRC bytes.

IRP.UserBuffer
Frame data — the contents of the frame.

Return Values

IoStatus.Status	IoStatus.Information
STATUS_BUFFER_TOO_SMALL	IO_ERR_TX_BUFFER_FULL
STATUS_INVALID_PARAMETER	IO_ERR_TX_FRAME_TOO_BIG

Refer also to the description of the interface record — the driver updates a field within the interface record reflecting the amount of buffer space available.

Remarks

In two-wire configurations, the driver must raise RTS and wait for CTS from the modem before initiating a transmission. The driver should then drop RTS when all frames have been transmitted. The driver assumes that the transmission is complete when both of the following are true:

- The transmit buffer becomes empty (if the link is configured as half-duplex).
- The last frame transmitted had the Poll/Final bit set in the second byte.

Function 0x45: Abort Transmitter

The SNALink calls this function to clear down the driver's transmitter.

Remarks

This request causes the driver to stop the current frame transmission and to flush its internal buffers of any further data pending transmission. In two-wire configurations, the driver should also drop RTS.

Function 0x46: Abort Receiver

The SNALink calls this function to clear down the driver's receiver.

Remarks

This request causes the driver to stop the receiver and to flush its internal buffers of any received data. It should be used to clear down the receiver, for instance, before altering the link characteristics.

Function 0x47: Off-Board Load

This function is not supported.

Function 0x61: Get/Set Interface Record

This function has been superseded by Function 0x64: Read Interface Record.

Function 0x62: Get V24 Status

The SNALink calls this function to read the current state of the modem interface lines. No parameter or data packet is passed. This request causes the driver to update the Input V.24 Status field in the driver interface record.

Function 0x63: Receive Frame

The SNALink calls this function to read a data frame from the driver's buffers.

Parameters

IRP.CurrentStackLocation.OutputBufferLength
Frame length — the size of the frame transferred into the data buffer by the driver. The frame includes the control, address, and information field (if present), but no flags or CRC bytes. When the request is issued, frame length is set to the maximum length of the buffer addressed by the data packet pointer.

IRP.UserBuffer
Frame data — the contents of the frame that has been received.

Return Values

IoStatus.Status	IoStatus.Information
STATUS_BUFFER_TOO_SMALL	None

Remarks

The driver transfers the next available received frame to the supplied buffer. Note that if the buffer is not large enough, a buffer overflow error is returned. This allows the application to decide if oversize frames are an error. If not, then a second attempt to read should be made, using a buffer at least Frame Length bytes long. A length of zero is returned if there are no frames ready to be received.

Function 0x64: Read Interface Record

This function reads the driver's interface record and copies it into the buffer passed by the SNALink. The buffer must be allocated by the SNALink prior to making this call.

Parameters

IRP.System.Buffer

Interface Record Address (IN) — a 32-bit pointer to the driver's interface record area.

The interface record format is as follows:

Description	Type
Received Frames	**int**
Transmit Buffer Space	**int**
Status Events	**int**
Input V.24 Status	**UCHAR**
Output V.24 Status	**UCHAR**
Statistics Counters	**int[11]**

- Received Frames is the number of frames currently queued in the driver receive buffers.
- Transmit Buffer Space is used to signal to the SNALink:

  Whether more frames can be provided to the driver.

  Whether the driver still has frames waiting to be sent.

  The Transmit Buffer Space field indicates the maximum frame size that the driver can currently accept. This is updated after each successful Transmit Frame IOCTL, and should be checked by the SNALink before sending further frames to the driver.

- Status Events is a count of the total number of increments made to the Statistics Counters.
- Input V.24 Status is a bitmap of the current logical state of the input interface lines, a value of 1 meaning ON and a value of 0 meaning OFF. The pins are mapped as follows:

Bit number	V.24 circuit name	Circuit number	RS-232C pin
7 - 5	Reserved	142	25
4	Test Indicator	125	22
3	Calling Indicator	125	22
2	RLSD (commonly DCD)	109	8
1	Data Set Ready (DSR)	107	6
0	Ready For Sending (CTS)	106	5

- Output V.24 Status is a bitmap of the current logical state of the output interface lines, a value of 1 meaning ON and a value of 0 meaning OFF. The pins are mapped as follows:

Bit number	V.24 circuit name	Circuit number	RS-232C pin
7 - 5	Reserved		
4	Maintenance Test	140	18
3	Select Standby	116	11
2	Data Signal Rate Selector	111	23
1	Data Terminal Ready (DTR)	108/2	22
0	Request to Send (RTS)	105	4

  Note that the driver will raise and lower RTS as necessary while transmitting, reflecting the state of RTS in the output V.24 status field. The application should not, therefore, try to manipulate RTS.

- The Statistics Counters are running counts of various kinds of link status information.The events they relate to are:

Counter number	Description
1	Frames received with incorrect CRC.
2	Frames larger than the maximum size received.
3	Frames smaller than 32 bits received.
4	Frames received that are not a multiple of 8 bits.

5	Aborted frames received.
6	Transmitter underruns.
7	Receiver overruns.
8	RLSD drops in mid-reception.
9	CTS drops in mid-transmission.
10	DSR drops.
11	Hardware failures (adapter or modem).

Remarks

The interface record provides a fast mechanism to use to decide whether a frame can be transmitted, whether there are any frames to be received, and so on. The driver maintains this information. Each time the SNALink requires this information, it calls **Read Interface Record** to get a copy of the current interface record. After each call, the driver clears the statistics counters in its own interface record. The V.24 status and transmit and receive data information are unchanged.

SNA Modem API

This section provides reference material for the SNA Modem API structure and functions.

MODEM_STATUS

A **MODEM_STATUS** structure for each SNA link contains status information used by the SNA Modem Status interface.

```
struct _ModemStatus{
    char LSName[12];
    char V24In;
    char V24Out;
    unsigned short RxFrameCount;
    unsigned short TxFrameCount;
    char Reserved[6];
} MODEM_STATUS;
```

Members

LSName
A character array containing the link service name.

V24In
A set of bit fields representing the V.24 input flags that determine which signal lines to mask. These bit fields can be **OR**ed together to create a complete mask. The defined bit fields for **V24In** are as follows:

MASK_CTS Mask the clear to send line.

MASK_DSR Mask the data set ready line.

MASK_DCD Mask the data carrier detect line.

MASK_DRI Mask the data ring indicator line.

V24Out
A set of bit fields representing the V.24 output flags that determine which signal lines to mask. These bit fields can be **OR**ed together to create a complete mask. The defined bit fields for **V24Out** are as follows:

MASK_RTS Mask the request to send line.

MASK_DTR Mask the data terminal ready line.

RxFrameCount
A count of received frames.

TxFrameCount
A count of transmitted frames.

Reserved
Padding for future expansion.

SNAModemInitialize

The **SNAModemInitialize** function should be called once per link service process at initialization. This function initializes the communication path to the SNA Modem application. The ideal place to call this function is in the **SNALinkInitialize** function.

```
void SNAModemInitialize();
```

See Also

SNALinkInitialize

SNAModemAddLink

The **SNAModemAddLink** function should be called once per link initialization. For link services that support more than a single SNA link, this call can be made multiple times. For link services that support only a single link, this call can be made immediately after **SNAModemInitialize**; otherwise it is preferable to call **SNAModemAddLink** as each port is initialized.

```
void SNAModemAddLink(
MODEM_STATUS **ppModemStatus
);
```

Parameters

ppModemStatus
The address of a pointer to a **MODEM_STATUS** structure that will be used for storing modem status information. The returned **MODEM_STATUS** structure will contain a link service name.

Remarks

The IHV should declare a pointer to a **MODEM_STATUS** structure and pass its address to **SNAModemAddLink**.

The **LSName** is initially the name of the link service, but may need to be altered for multiport link services.The IHV can replace the link service name returned in the **MODEM_STATUS** structure to differentiate between possible multiple connections through a single link service.

The IHV should maintain the various input and output signal lines and the data flow frame counts in the returned **MODEM_STATUS** structure. The Host Integration Server 2000 Modem Monitor application will periodically read and display the data stored in this **MODEM_STATUS** structure.

Internally **SNAModemAddLink** increments the usage count of the shared memory, and signals the Host Integration Server 2000 Modem Monitor application that a new link has been added.

See Also

MODEM_STATUS, SNAModemInitialize

SNAModemDeleteLink

The **SNAModemDeleteLink** function should be called when a link is terminating to delete the resources associated with the link. The parameters passed in must correspond to those returned by a call to **SNAModemAddLink**.

```
void SNAModemDeleteLink(
MODEM_STATUS *pModemStatus
);
```

Parameters

pModemStatus
A pointer to a **MODEM_STATUS** structure that was passed to the **SNAModemAddLink** function used for storing modem status interface.

Remarks

All resources in the link service associated with the modem status are deleted, and they must not be accessed by the IHV code after calling this function.

See Also

MODEM_STATUS, SNAModemAddLink

SNAModemTerminate

The **SNAModemTerminate** function should be called once per link service process, at termination. The ideal place is **SNALinkTerminate**. If the link service supports a single link, it is appropriate to call **SNAModemDeleteLink** immediately before **SNAModemTerminate**. Otherwise it is better to call **SNAModemDeleteLink** as each link instance is terminated.

```
void SNAModemTerminate();
```

See Also

SNALinkTerminate, SNAModemDeleteLink

SNA Perfmon API

This section provides reference material for the SNA performance monitoring structures and functions.

ADAPTERCOUNTER

The **ADAPTERCOUNTER** structure represents an individual SNA Perfmon event that can be monitored, such as the total bytes transmitted. All of the data needed to display a single Perfmon event is stored in this structure.

```
typedef struct adaptercounter
{
    ULONG count;
    ULONG type;
    LONG scale;
} ADAPTERCOUNTER;
```

Members

count
The count for a specific Perfmon event since startup of the link service. Each time a Perfmon event takes place, the count is incremented accordingly, based on the type of event being counted. This count is maintained by the link service.

type
The event type that is being monitored with this **ADAPTERCOUNTER**. The **type** member instructs Perfmon whether the **count** member represents a numeric counter such as number of connection failures, a rate such as throughput in bytes transferred per second, or a percentage. For suitable values see Platform SDK documentation of **PERF_COUNTER_*** (for example, **PERF_COUNTER_COUNTER** or **PERF_COUNTER_RAWCOUNT**).

scale
The default scale to be used by the Perfmon application when displaying this event. The **count** member is scaled by 10**scale** such that a **scale** of -1 multiplies **count** by 0.1.

ADAPTERPERFDATA

The **ADAPTERPERFDATA** structure groups all of the ADAPTERCOUNTER structures for an SNA link service together into a single block. It also has a few fields used internally by the SNA Perfmon code. The SNA link driver should not change the first three members of this structure.

```
typedef struct adapterperfdata
{
    ULONG inuse;
    ULONG ServiceNameIndex;
    ULONG FirstCounterIndex;
    ADAPTERCOUNTER TotalBytesReceived;
    ADAPTERCOUNTER TotalBytesTransmitted;
    ADAPTERCOUNTER TotalFramesReceived;
    ADAPTERCOUNTER TotalFramesTransmitted;
    ADAPTERCOUNTER SuccessfulConnects;
    ADAPTERCOUNTER ConnectionFailures;
    ADAPTERCOUNTER TotalBytesThroughput;
    ADAPTERCOUNTER TotalFramesThroughput;
    ADAPTERCOUNTER AdapterFailures;
    ADAPTERCOUNTER reserved[11];
    ULONG pad;
} ADAPTERPERFDATA;
```

Members

inuse
A flag that indicates that the link service is using this section of shared memory.

ServiceNameIndex
An index into an array of strings describing events that can be monitored by the Perfmon functions. These strings are stored in the registry under the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib** key.

FirstCounterIndex
An index into an array of events that can be monitored by the Perfmon functions.

TotalBytesReceived
The number of data bytes received per second.

TotalBytesTransmitted
The number of data bytes transmitted per second.

TotalFramesReceived
The number of data frames received per second. A frame is an information structure recognized by one of the various protocols related to SNA. Frames contain multiple bytes of data.

TotalFramesTransmitted
The number of data frames transmitted per second.

SuccessfulConnects
The number of times since startup that a successful connection has been made.

ConnectionFailures
The number of times since startup that a connection has encountered an error condition.

TotalBytesThroughput
The total number of bytes flowing through the Host Integration Server 2000 per second. This includes both incoming and outgoing bytes, and is a good indicator of how heavily your Host Integration Server 2000 is loaded.

TotalFramesThroughput
The total number of data frames flowing through the Host Integration Server 2000 per second. This includes both incoming and outgoing frames, and is a good indicator of how heavily your Host Integration Server 2000 is loaded.

AdapterFailures
The number of times since startup that a network adapter has encountered an error condition.

reserved
An array of **ADAPTERCOUNTER** structures for future expansion.

pad
Padding.

SNAInitLinkPerfmon

The **SNAInitLinkPerfmon** function initializes the Perfmon data structures and code for an SNA link. The user defines the address of a handle and a void pointer that are passed in as parameters to this function. This function returns values in these parameters that are then used by subsequent calls to the Perfmon code. The SNA link driver should not modify the parameters returned by this function.

```
void SNAInitLinkPerfmon(
    HANDLE *shrlockmutx,
    void **shrptr
);
```

Parameters

shrlockmutx
An address of a handle of a mutex used to protect a block of shared memory. This handle address is used when calling other Perfmon functions after initialization.

shrptr
The address of a pointer to a block of shared memory used by subsequent Perfmon functions.

SNAGetLinkPerfArea

The **SNAGetLinkPerfArea** function returns a pointer to the shared data area used by the Perfmon application to store the link statistics. The parameters are the returned values from **SNAInitLinkPerfmon**. The SNA link then maintains the **ADAPTERCOUNTER** members of the returned **ADAPTERPERFDATA** structure.

```
ADAPTERPERFDATA * SNAGetLinkPerfArea(
HANDLE shrlockmutx,
ADAPTERPERFDATA *shrptr
);
```

Parameters

shrlockmutx
The handle of a mutex used to protect a block of shared memory that will contain the **ADAPTERPERFDATA** structure for this SNA link. The address of this handle is returned by the **SNAInitLinkPerfmon** function.

shrptr
A pointer to a block of shared memory returned by **SNAInitLinkPerfmon** that will contain the **ADAPTERPERFDATA** structure used by the Perfmon functions for this SNA link.

See Also

ADAPTERPERFDATA, SNAInitLinkPerfmon

SNAGetPerfValues

The **SNAGetPerfValues** function is used to provide pointers to the *ServiceNameIndex* and *FirstCounterIndex* variables so that the Perfmon application knows where to get the descriptions of the performance counters from the registry. These variables are returned as members in the **ADAPTERPERFDATA** structure returned by the **SNAGetLinkPerfArea** function.

```
USHORT SNAGetPerfValues(
int *pServiceNameIndex,
int *pFirstCounterIndex
);
```

Parameters

pServiceNameIndex
A pointer to the **ServiceNameIndex** member of the **ADAPTERPERFDATA** structure.
pFirstCounterIndex
A pointer to the **FirstCounterIndex** member of the **ADAPTERPERFDATA** structure.

See Also

ADAPTERPERFDATA, SNAGetLinkPerfArea

Administration and Management Programming

This section of the Microsoft® Host Integration Server 2000 Developer's Guide describes how to use Windows® Management Instrumentation (WMI) to administer, manage, monitor status, and collect performance information about Microsoft Host Integration Server 2000.

This section contains:

Introduction to Host Integration Server Administration with WMI

This section of the *Microsoft Host Integration Server 2000 Developer's Guide* provides information required to develop applications using the Microsoft® Windows® Management Instrumentation (WMI) to configure, manage, monitor status, and collect performance information about Microsoft Host Integration Server 2000.

The Windows Management Instrumentation (WMI) technology is an implementation of the Desktop Management Task Force's (DMTF) Web-Based Enterprise Management (WBEM) initiative for Microsoft Windows platforms that extends the Common Information Model (CIM) to represent management objects in Windows management environments. The Common Information Model, also a DMTF standard, is an extensible data model for logically organizing management objects in a consistent, unified manner in a managed environment.

Based on the Common Information Model, WBEM is a DMTF initiative and technology that provides a standardized way to access management information in an enterprise environment. With WBEM, developers can create tools and technologies that reduce the complexity and costs of enterprise management. By providing such standards, WBEM contributes to the industry-wide efforts to lower Total Cost of Ownership (TCO). TCO refers to the administrative costs associated with computer hardware and software purchases, deployment and configuration, hardware and software updates, training, maintenance, and technical support.

WBEM provides a point of integration through which data from management sources can be accessed, and it complements and extends existing management protocols and instrumentation such as Simple Network Management Protocol (SNMP), Desktop Management Interface (DMI), and Common Management Information Protocol (CMIP).

The WBEM initiative results from the cooperative efforts of BMC Software Inc., Cisco Systems Inc., Compaq Computer Corp., Intel Corp. and Microsoft Corp., as well as many other companies active in the DMTF.

The Windows Management Instrumentation (WMI) technology, Microsoft's implementation of WBEM, is a management infrastructure that supports the syntax of CIM, the Managed Object Format (MOF), and a common programming interface. The Managed Object Format is used to define the structure and contents of the CIM schema in human and machine-readable form. Windows Management Instrumentation offers a powerful set of services, including query-based information retrieval and event notification. These services and the management data are accessed through a Component Object Model (COM) programming interface. The WMI scripting interface also provides scripting support.

The WMI technology when used with Host Integration Server 2000 provides:

- Access to monitor, command, and control Host Integration Server 2000 through a common, unifying set of interfaces, regardless of the underlying instrumentation mechanism. WMI is an access mechanism.
- A consistent model of Host Integration Server 2000 operation, configuration, and status.
- A COM Application Programming Interface (API) that supplies a single point of access for all management information.
- Interoperability with other Windows 2000 management services. This approach can simplify the process of creating integrated, well-designed management solutions.
- A flexible, extensible architecture. Developers can extend the information model to cover new devices and applications by writing code modules called WMI providers.
- Extensions to the Windows Driver Model (WDM) to capture instrumentation data and events from Host Integration Server 2000 device drivers and kernel side components.
- A powerful event architecture. This allows management information changes to be identified, aggregated, compared, and associated with other management information. These changes can also be forwarded to local or remote management applications.
- A rich query language that enables detailed queries of the information model.
- A scriptable API which developers can use to create management applications. The scripting API supports several languages, including Microsoft® Visual Basic®; Visual Basic for Applications; Visual Basic, Scripting Edition (VBScript); Microsoft® JScript® development software. Additionally, you can use the Windows Script Host or Microsoft Internet Explorer to write scripts utilizing this interface. Windows Script Host, like Internet Explorer, serves as a controller engine of ActiveX Scripting engines. Windows Script Host supports scripts written in VBScript, and JScript.

Windows Management Instrumentation (WMI) is the name given to Microsoft's implementation of WBEM. On Microsoft Windows 2000, WMI is the standard way to expose management functions to application programmers.

Administration and management applications used in a Host Integration Server 2000 environment can be developed using several different development tools and application programming interfaces including:

- C or C++ applications that use WMI COM APIs.
- C or C++ applications that use ODBC to access WMI.
- Microsoft Visual Basic® applications that use ActiveX® Data Objects (ADO) to access WMI.
- Microsoft Visual Basic for Applications to access WMI.

In addition, administration and management applications can be developed using several different scripting tools and application programming interfaces including:

- Microsoft Visual Basic Scripting Edition (VBScript).
- Microsoft JScript®.
- VBScript or JScript written as Microsoft Active Server Pages (ASP) running in conjunction with Microsoft Internet Information Server (IIS) and web-based clients.
- WMI Query Language (WQL) applications.
- Windows Host Script (WSH) scripts that access WMI.

Additionally, you can also use any scripting language implementation that supports Microsoft's ActiveScripting technologies with this API such as a Perl scripting engine.

The WMI Query Language is a subset of standard SQL designed specifically to access management information using WMI.

To use this guide effectively, you should be familiar with:

- Microsoft Host Integration Server 2000
- SNA concepts
- Microsoft Windows Management Instrumentation (WMI)
- One of the following operating environments:
    - Microsoft Windows 2000
    - Microsoft Windows NT®
    - Microsoft Windows XP
    - Microsoft Windows Millennium Edition
    - Microsoft Windows 98
    - Microsoft Windows 98
    - Microsoft Windows 95

Depending on the application programming interface and development tools used, you should be familiar with:

- WMI COM/DCOM APIs
- WMI Query Language
- WMI schema and MOF file syntax
- Microsoft Windows Script Host
- Microsoft Visual Basic
- Microsoft Visual Basic for Applications
- Microsoft Visual Basic Scripting Edition
- Microsoft JScript
- Microsoft ODBC
- Microsoft ADO
- Microsoft Active Server Pages

For more information about WMI, see *Windows Management Instrumentation (WMI)* in the Microsoft Developer Network (MSDN®) Platform Software Development Kit.

Administration Programmer's Guide

This section of the Microsoft® Host Integration Server 2000 Developer's Guide describes the programmatic techniques and procedures for using WMI in conjunction with Microsoft Host Integration Server 2000.

This section contains:

WMI and Host Integration Server Architecture

The architecture of the Windows Management Instrumentation (WMI) technology consists of the following components:

- Management applications
- Managed objects
- Providers
- Management infrastructure, consisting of the WMI software (Winmgmt.exe) and the CIM repository

Management applications are Microsoft® Windows®-based applications or services on Microsoft Windows NT® or Windows 2000 that process or display data from managed objects. A management application can perform a variety of tasks in a Host Integration Server environment such as configuring HIS Servers, measuring performance, reporting outages, and correlating data.

Managed objects represent logical or physical enterprise components. Managed objects are modeled using the CIM, and they are accessed by management applications through WMI. A managed object in the Microsoft Host Integration Server 2000 environment can be any component of the system, from a link service device driver communicating with hardware to software configuration information on users and connected Logical Units (LUs).

The WMI providers supplied with Host Integration Server 2000 use the WMI COM API to supply the WMI repository with data from Host Integration Server managed objects, to handle requests on behalf of Host Integration Server management applications, and to generate notifications of events. The WMI providers included with Host Integration Server 2000 include the following:

- Host Integration Server configuration provider (wmiHIS)
- Host Integration Server MSMQ-MQseries Bridge provider (wmiMQBridge)
- Host Integration Server SNA configuration provider (wmisna)
- Host Integration Server SNA status provider (wmisnastatus)
- Host Integration Server SNA trace provider (wmisnatrace)

In addition, Windows 2000 includes several standard WMI providers, such as a registry provider, for accessing information from the system registry. Windows 2000 also supplies a Windows 2000 Event Log Provider that allows applications to receive notifications of Windows 2000 and Host Integration Server 2000 events and to access the information stored in the Windows 2000 event log. The Win32® WMI providers are also available for use with Windows NT 4.0 (Service Pack 5 or later) as part of the WMI Software Development Kit (SDK).Third-party vendors can create custom providers to interact with managed objects specific to their environment.

The management infrastructure consists of WMI and the CIM repository. WMI enables users to handle communications between management applications and providers. Users store their static data in the CIM repository. Applications and providers communicate through WMI using a common programming interface (COM API). The COM API, which supplies event notification and query processing services, is available in the C and C++ programming languages.

The CIM repository holds static management data. Static data is data that does not regularly change. WMI also supports dynamic data, which is data that must be generated on demand because it is frequently changing. Data can be placed in the CIM repository by WMI or network administrators. Information can be placed in the CIM repository using either the Managed Object Format (MOF) language and the MOF Compiler or the WMI COM APIs. The WMI providers supplied with Host Integration Server 2000 use both mechanisms.

The following MOF files are supplied with Host Integration Server 2000 for use by management applications:

- wmiHIS.mof
- wmiMQbridge.mof
- wmisna.mof
- wmisnastatus.mof
- wmisnatrace.mof

Management applications can access the COM API directly to interact with WMI and the CIM repository to make management requests of Host Integration Server 2000. Applications can also use other access methods such as Open Database Connectivity (ODBC) and the Hypertext Markup Language (HTML) to make these requests. An ODBC Driver for WMI is included with Windows 2000. The protocol used for communication between local and remote components is Distributed Component Object Model (DCOM).

Platforms Supported by WMI and Host Integration Server

Windows Management Interface (WMI) is included as part of Microsoft® Windows® 2000, Windows XP, and Windows Millennium Edition. On Microsoft Windows NT® 4.0, Windows 98, and Windows 95, WMI must be installed as part of the WMI Software Developers Kit (SDK) which is available for download from the Microsoft Download Center: http://go.microsoft.com/fwlink/?LinkId=12753.

This will bring up the Microsoft Downloads search page.

1. Select "Keyword Search" radio button.
2. Enter "WMI" into the "Keywords" edit box.
3. Select the desired target operating system.
4. Select the "All Downloads" in the "Show Results for" dropdown list
5. Click the "Find It" button.
6. Download the file by clicking on its link.
7. Install WMI by running wmisdk.EXE (double-click the file from your Windows Explorer). This will create directories for WMI in the system folder.

The WMI SDK is also included as part of the Microsoft Developer Network (MSDN®) Platform SDK.

For use with Host Integration Server, the WMI SDK version 1.1 or later can be installed on one of the following operating systems:

- Microsoft Windows 2000 Server
- Microsoft Windows 2000 Advanced Server
- Microsoft Windows 2000 Datacenter Server
- Microsoft Windows 2000 Professional
- Microsoft Windows NT Server 4.0 with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Enterprise Edition with Service Pack 6a or later
- Microsoft Windows NT Server 4.0, Terminal Server Edition with Service Pack 6a or later
- Microsoft Windows NT Workstation 4.0 with Service Pack 6a or later
- Microsoft Windows 98, Second Edition

Host Integration Server 2000 with Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

Windows Script Host (WSH) is installed by default with Windows 2000. For use on Windows NT 4.0, Windows 98, and Windows 95, Windows Script can be downloaded from the following Web site: http://go.microsoft.com/fwlink/?LinkId=13285.

The Windows Script download includes Microsoft Visual Basic® Scripting Edition (VBScript), Microsoft JScript®, Windows Script Components, Windows Script Host, and Windows Script runtime. Please follow the download and setup instructions on this site for Windows Script Host. Complete information about Windows Script Host is available as part of the Windows 2000 MSDN Platform SDK.

Programming Considerations Using WMI and Host Integration Server

Microsoft® Host Integration Server 2000 provides several WMI providers. The MOF files used by Host Integration Server are installed in the System directory below where Host Integration Server is installed. The default location for these files would be in the following subdirectory:

```
C:\Program Files\Host Integration Server\System
```

To use or view these MOF files on other computers (a computer with the Administration client or End-user Client, for example) to develop applications, these MOF files must be copied from the Host Integration Server computer. These MOF files document the WMI providers supplied with Host Integration Server and the CLSIDs, classes, properties, and methods supported by these providers .

Security using Active Server Pages (ASP) is handled differently on Microsoft Windows NT® 4.0 and Microsoft Windows® 2000 platforms. This affects WMI Scripting permissions for ASP scripts run on Internet Information Server. These permissions will affect the operation of the WMI ASP samples included with the Host Integration Server SDK as well as any other applications developed using WMI and ASP.

On Windows NT 4.0, the following registry entry under HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft enables or disables running ASP scripting with WMI:

WBEM\ScriptingKey=Enable for ASP

This entry is a REG_DWORD where the value 0x0 turns this option off (disables) and the value of 0x1 turns this option on (enables). This registry key only applies to Windows NT 4.0. WMI scripting using Active Server Pages is enabled automatically on Windows 2000.

For the proper security setting for ASP on Windows 2000, it is recommended that you set Anonymous Authentication off and enable Integrated Windows Authentication in the IIS Server configuration for directories with ASP files used with Host Integration Server. In order to access Host Integration Server configuration and status information, an application or user must have the appropriate administrative rights which are not available with anonymous authentication. The following procedure can be used on Windows 2000 to properly configure security using Active Server Pages:

To configure security using Active Server Pages

1. Open the Internet Information Services (can be found under the Administrative tools under the Start Button or the Control Panels).
2. Find the directory where the ASP files reside.
3. Right click the directory and select the Properties option.
4. When the next dialog appears, select the Directory Security Tab.
5. Click the Edit button in the Anonymous Authentication frame.
6. When the next dialog opens, uncheck the checkbox next to Anonymous Authentication.
7. Check the checkbox next to Integrated Windows Authentication.
8. Click the OK buttons to save these settings.

This will set that particular directory to use Integrated Windows Authentication as opposed to Anonymous Authentication without affecting any of your other directories. If there are any other ASP files that require or allow Anonymous Authentication you might want to make a new directory that you can turn off Anonymous Authentication and put you WMI ASPs there. Any script that calls ExecMethod from a Active Server Page should be setup to use Integrated Windows Authentication to verify the user attempting to run the script.

In addition, when using a "REFRESH" variable on a Web page and the page is being used to start and stop snaservices via ASP scripting, the Web browser client (Internet Explorer, for example) should set the following option:

To use a Web browser client to start and stop snaservices via ASP scripting

1. In Internet Options, under the General Tab, select Settings.
2. Ensure that "Every visit to the page" is selected in the "Check for newer versions of Stored pages:" option.

If this change is not made on the Web browser client, some ASP scripts will not run properly due to Internet Explorer caching

older results.

There are some restrictions on using WMI and Host Integration Server 2000 regarding connections to backup servers. The Snabase works to synchronize the information in the COM.CFG configuration file on the primary server across all backup servers. Each backup server has a local copy of the COM.CFG file from this synchronization process. WMI has a limitation that it will not attempt to read the local backup server's copy of COM.CFG if the primary server is alive. This request will always be forwarded to the primary server. A client that connects to a WMI provider running on a backup server will not be able to either retrieve any information or make configuration changes. This is a limitation of DCOM which does not permit impersonation outside the local machine. When a client on one computer connects to a WMISNA provider on another computer that is a backup server, the client is using DCOM to connect to the backup Host Integration Server computer. When the WMI provider on the backup server tries to access the COM.CFG file from the primary server, this is not allowed by DCOM because the application is trying to impersonate the user across the machine boundary. It might be possible to work around this limitation on Windows 2000 using delegation, but it is not possible on Windows NT 4.0 where delegation is not supported. Consequently, these replicated copies are used if and only if the primary server is down, in which case it'll fail-over to a backup and this limitation will not apply.

An ImportExport sample program written in Microsoft Visual Basic® Scripting Edition (VBScript) is provided as part of the Host Integration Server SDK. This tool allows configuration information from Host Integration Server to be exported and saved to a text file using WMI in MOF format. This text file can also be changed and imported using this sample program to change configuration information.

A potential problem using WMI can occur with duplicate LU pools that can be illustrated using this sample program. Normally, exporting and re-importing the MOF file would not create duplicates. However, the Host Integration Server WMI provider allows pool to workstation association instances to be duplicated because, by design, duplicates of this type of object are allowed. It is possible to associate the same pool to the same workstation or user multiple times. This is used by emulators to create more sessions for clients. Therefore, it is not possible to identify one such association from another. The WMISNA Provider, WMISNA.DLL, will always create new associations of these types, even if an association with the same pair (Pool, Wks) already exists. This object type is allowed only in this specific case. However, this can create a problem for applications developed using WMI (the Import/Export sample, for example) if the application doesn't know not to create the duplicates.

The follow sequence illustrates this issue using the ImportExport sample:

1. Using SNA Manager to create a pool workstation association.
2. Export the SNA configuration to a MOF file using the ImportExport utility.
3. Import that same MOF file again using the ImportExport utility.
4. Duplicate pool-workstation associations will be created.

The result is that if a client uses the import/export sample or a similar application developed using WMI on a Host Integration Server configuration that has Pool to Workstation associations, then the number of associations will effectively double after running the sample. The workaround using the ImportExport sample would be as follows:

1. Export the configuration to a MOF file.
2. Remove the pool to workstation associations from the MOF file just created.
3. Import the MOF file back.

When importing the configuration from one domain to another using the ImportExport sample or a similar application developed using WMI, then step 2 should be ignored. Normally, WMI applications should copy an existing configuration to a blank configuration file so this condition does not arise.

Administration Sample Applications

The source code for several sample programs that illustrate using Windows® Management Instrumentation (WMI) for administration and management of Host Integration Server are included on the Microsoft® Host Integration Server 2000 CD-ROM and as part of the Microsoft Developer Network (MSDN) Platform SDK. These sample programs are located in the \SDK\Samples\Admin subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the Samples\Admin subdirectory below where the Host Integration Server SDK is installed (C:\Program Files\Host Integration Server SDK, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the Samples\NetDS\HIS\Admin subdirectory below where the MSDN Platform SDK has been installed (C:\Program Files\Microsoft SDK, by default).

These sample programs include the files in the following subdirectories:

File or Subdirectory	Description
ImportExport	A WMI sample script written in Microsoft® VBScript that illustrates how to import and export configuration information from Host Integration Server.
SNAWMI.ASP	The main page of a WMI sample script written in Microsoft® Active Server Pages (ASP) for retrieving configuration information from Host Integration Server using WMI.
SnaWebAdmin	Subsidiary pages of WMI sample scripts written in Microsoft® Active Server Pages (ASP) for retrieving configuration information from Host Integration Server using WMI. Each one of these subdirectories contains ASP sample scripts that illustrate how to retrieve information from Host Integration Server on a specific feature.

Several sample programs with source code are provided with Host Integration Server 2000 that illustrate administration and management.

This section contains:

- Active Server Pages SNAWebAdmin sample programs
- VBScript ImportExport sample program

Active Server Pages SNAWebAdmin Sample

The Admin\SnaWebAdmin folder contains a collection of Active Server Pages (ASP) for use with a Web server application designed to access configuration, management, and status information from the SNA server component of Microsoft® Host Integration Server. These sample applications require Microsoft Internet Information Server version 3.0 or higher with Active Server Pages be installed. Host Integration Server 2000 and Internet Information Server must be installed and running on the same machine.

The WMI ASP samples must be installed into the Web server's public directories below WWWRoot. Copy the contents of the entire Admin directory from the SDK\Samples\Admin subdirectory including SNAWebAdmin subdirectory to your WWWROOT directory on the Web server. After these files have been copied you should have a copy of the SNAWMI.ASP, fphover.class, and fphoverx,class files in WWWROOT and a WWWROOT\SNAWebAdmin folder with a series of subdirectories containing a number of ASP and GIF files.

The samples may then be run by opening Internet Explorer or some other Web browser on the same machine or a different machine and entering the following URL in the address line:

```
http://<machine name>/SNAWMI.asp
```

Substitute the network name of the machine hosting the Web server and Host Integration server for the machine name in angle brackets in the URL above. This will open the main page of the SNAWebAdmin ASP application and allow you to select any of the other sample ASP pages. Information about each sample is provided on this web page. Additional information is available on the following URL:

```
http://<machine name>/admin/headers/welcome.htm
```

These ASP pages illustrate using WMI to retrieve SNA management and configuration from Host Integration Server.

The two Java class files, fphover.class and fphoverx.class, are redistributable files that ship with Microsoft FrontPage®. These Java class files are used in some of the WMI sample scripts instead of a submit button to stop and start services.

Several subdirectories below SNAWebAdmin need to have IIS security enabled (no anonymous access). Otherwise the scripts in these subdirectories will fail since the anonymous user account by default does not have access rights that would allow it to start or stop services on Microsoft Windows NT® or Windows® 2000 or make changes to the Host Integration Server system. The subdirectories that need IIS security enabled are the following:

```
SNASebAdmin\Change
SNASebAdmin\Connections
SNASebAdmin\Services
SNASebAdmin\Status
```

It is possible to host these ASP pages on a machine running the Web server that is different from the machine running Host Integration Server. However, this requires some changes to the ASP pages to handle connections to a different machine, security, and authentication issues.

VBScript ImportExport Sample

The Admin\ImportExport folder contains a sample written in Microsoft® Visual Basic® Scripting Edition (VBScript) that illustrates how to import and export SNA configuration information from Host Integration Server in MOF format using WMI. This sample relies on the MOFCOMP.exe application supplied with Microsoft Windows® 2000 or installed as part of WMI 1.1 on Microsoft Windows NT® 4.0 for importing.

In the examples below, ImportExport.VBS has been renamed to HISCFG.vbs.

The usage for this command-line tool for exporting configuration information is as follows:

```
HISCFG [/S:server] [/N:namespace] [/C:class] [/O:outfile]
       [/U:username] [/W:password] [/Q]
```

The various command-line switches are explained in the table below. Note that case is ignored for command line options except for help and either the '/' or '-' character is interpreted as the leading character for an option. The table below uses the '/' character for illustration.

Command-Line Switch	Comments
/?	This flag shows the usage for this command and exits.
/C	The name of the WMI parent class to be queried. This should be set to one of the WMI classes defined in the MOF files supplied with Host Integration Server. This parameter defaults to MsSna_Config and exports all of the classes SNA classes and their associations.
/h	This flag shows the usage for this command and exits.
/N	The WMI namespace to be queried. This parameter defaults to "root\MicrosoftHIS" for Host Integration Server.
/O	The name of the file used for output. This parameter has no default value.
/Q	This Boolean flag indicates whether this is query should be completed quietly without displaying any status or error messages. This parameter defaults to verbose option.
/S	The name of the machine running Host Integration Server. This parameter has no default value.
/U	The user name of a user on the domain or active directory where Host Integration Server is running with administrative rights. This parameter has no default value.
/W	The password of a user on the domain or active directory where Host Integration Server is running with administrative rights. This parameter has no default value.

The usage for this command-line tool for importing SNA configuration information is as follows:

```
HISCFG [/I:inputfilename]
```

A potential problem using WMI can occur with duplicate LU pools that can be illustrated using this sample program. Normally, exporting and re-importing the MOF file would not create duplicates. However, the Host Integration Server WMI provider allows pool to workstation association instances to be duplicated because, by design, duplicates of this type of object are allowed. It is possible to associate the same pool to the same workstation or user multiple times. This is used by emulators to create more sessions for clients. Therefore, it is not possible to identify one such association from another. The WMISNA Provider, WMISNA.DLL, will always create new associations of these types, even if an association with the same pair (Pool, Wks) already exists. Only in the case of this object type is this allowed. However, this can create a problem for applications developed using WMI (the Import/Export sample, for example) if the application doesn't know to not create the duplicates.

The follow sequence illustrates this issue using the ImportExport sample:

1. Using Host Integration Server Manager or the Administration Manager client create a Pool-Workstation association.
2. Export the SNA configuration to a MOF file using the ImportExport utility.
3. Import that same MOF file again using the ImportExport utility.
4. Duplicate Pool-Workstation associations will be created.

   The result is that if a client uses the import/export sample or a similar application developed using WMI on a Host Integration Server configuration that has Pool to Workstation associations, then the number of associations will effectively

double after running the sample. The workaround using the ImportExport sample would be as follows:

5. Export the configuration to a MOF file.
6. Remove the pool to workstation associations from the MOF file just created.
7. Import the MOF file back.

When importing the configuration from one domain to another using the ImportExport sample or a similar application developed using WMI, then step 2 should be ignored. Normally, WMI applications should copy an existing configuration to a blank configuration file so this condition does not arise.

Client Binary Setup

This section provides information regarding the installation of the Microsoft® Host Integration Server 2000 and Microsoft SNA Server 4.0 client binaries. This information will enable you to consolidate Host Integration Server or SNA Server client binaries into your product's own setup, simplifying operation for the end user.

The Host Integration Server 2000 client binaries are installed by using the Microsoft System Installer (MSI) and MSI package files.

A Microsoft Windows® installer package (.msi) file is a storage file containing the instructions and data required to install an application. The MSI packages included with Host Integration Server 2000 can be used as part of the installation procedure for your products. Windows 2000 includes the MSI runtime required to install the MSI packages supplied with Host Integration Server 2000. On Microsoft Windows NT® 4.0, Windows 98, and Windows 95, the MSI runtime must be installed prior to using the Host Integration Server 2000 MSI packages.

Host Integration Server 2000 with Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

Windows XP includes the MSI runtime required to install the MSI packages with Host Integration Server 2000.

If you wish to modify or create a custom MSI package integrating your product and the Host Integration Server client binaries, then you will need to create new MSI packages. The MSI packages supplied with Host Integration Server 2000 were constructed using the WISE system installer and can only be modified by using this product. Because of differences in tools, it is not possible to easily modify an MSI package created using the WISE system installer using other products such as InstallShield 2000.

Documentation on creating an MSI package or using Microsoft System Installer tools is not provided with Host Integration Server 2000. The Microsoft System Installer is documented as part of the Microsoft Software Development Network (MSDN®) Platform Software Development Kit (SDK).

The earlier SNA Server 4.0 and SNA Server 3.0 client binaries were installed using the Microsoft Setup Toolkit (ACME Setup Toolkit). The ACME Setup Toolkit is completely different from setup procedures using the Microsoft System Installer and documentation on its use is not provided with Host Integration Server 2000 and SNA Server. Note that in releases of SNA Server earlier than version 3.0, the client binaries were installed on Microsoft Windows NT and Microsoft Windows 95 using the Windows NT Setup engine and on Windows 3.x using the Windows setup tools.

This document assumes that your own product will be installed using a third-party installation tool, although it is also possible to use the Microsoft System Installer tools, the Windows NT Setup engine, or the Windows 3.x setup tools. Installation information is provided for the following environments:

- Windows XP, Windows 2000, Windows NT, Windows Millennium Edition, Windows 98, and Windows 95 (Win32®)
- Windows 3.x (16-bit Windows)

This information should allow you to integrate the installation of the Host Integration Server 2000 or SNA Server client binaries with your own product.

This section contains:

Client Setup for Windows 2000, Windows NT, Windows 98, and Windows 95

When installing Microsoft® Host Integration Server 2000 client binaries in the Win32® environments of Microsoft Windows® XP, Windows 2000, Windows NT®, Windows Millennium Edition, Windows 98, or Windows 95, the Microsoft System Installer (MSI) packages supplied with Host Integration Server 2000 should be used. These supplied MSI packages can be invoked as part of your custom installation procedure to install client binaries.

The MSI Client packages are located on the Host Integration Server 2000 CD-ROM in separate subdirectories under the Setup\Clients folder. The client MSI packages supplied with Host Integration Server 2000 are as follows:

MSI Package File	Description
Administrator\HIAdmin.msi	Administrator client for use on Windows XP Professional, Windows 2000, and Windows NT
EndUser\HIClient.msi	End-user client for use on Windows XP (Professional or Home Edition),Windows 2000, Windows NT, Windows 98, and Windows 95.

A template INF file is supplied with each of these MSI packages that lists all possible client binary setup options available in each package and the default option that will be selected by default. The Host Integration Server 2000 MSI packages and the MSIEXEC.exe tool supplied as part of the Microsoft System Installer support a batch mode setup option that can be used with a modified version of these INF files configured with your preferred setup options.

The Host Integration Server 2000 CD-ROM also contains a Setup\Clients\Web folder containing the files needed for a Web-based installation of the SNA 3270 and 5250 clients to a client computer using a Web browser across a corporate intranet. No end-user configuration is required. Users click a hyperlink and connect to their host application through Host Integration Server 2000. The Host Integration Server 2000 web clients are compatible with Microsoft Internet Explorer version 3.02 or higher and can be used on the Microsoft Windows XP, Windows 2000, Windows NT version 4.0, Windows 98, and Windows 95 operating systems.

The following sections provides information regarding installation of the Microsoft SNA Server 4.0 and 3.0 Client binaries in the Win32 environments of Microsoft Windows NT and Windows 95. This information will enable you to consolidate SNA Server into your product's own setup, simplifying operation for the end user. SNA Server versions 3.0 and later support SNA Server versions 2.0 and 2.1 Client binaries for Win32 environments, although it is recommended that you release updated binaries with each normal product release.

This section describes the use of SNA Server versions 3.0 and later binaries, which are recommended for all new applications.

This section contains:

SNA Server Client Binary Files for Win32 Environments

The binary files on the SNA Server 4.0 and 3.0 distribution CD-ROM for use on Windows 2000 and Windows NT are found in the \CLIENTS\WINNT and \CLIENTS\WINNT\SYSTEM directories, and the binary files for use on Windows 98 and Windows 95 are found in the \CLIENTS\WIN95 and \CLIENTS\WIN95\SYSTEM directories. You will need at a minimum to distribute the following files:

File name	Purpose
ADLOC.DLL	Locates SNA Servers.
APPCST32.DLL	Translates APPC return codes into text strings.
ASYNCTRC.DLL	Asynchronous trace DLL (Windows 95 and Windows 98 only).
COMTBLG.DAT	Table for type G character set conversion (ASCII <–>EBCDIC).
CSVSTR32.DLL	Translates Common Service Verb (CSV) return codes into text strings.
DBGTRACE.DLL	Trace DLL (asynchronous).
DMODAPPC.DLL	A support DLL used by SNAKRNL.DLL to provide a network protocol-independent transport interface for SNA Server Client applications (used only on Windows 95 and Windows 98).
FMISTR32.DLL	Translates EIS/LUA error codes into text strings.
MFC40.DLL	MFC run-time support.
MFC40U.DLL	MFC run-time support.
MSVCIRT.DLL	Visual C/C++ run-time support.
MSVCRT.DLL	Visual C/C++ run-time support.
MSVCRT40.DLL	Visual C/C++ run-time support.
OLEPRO32,DLL	OLE run-time support.
SNABASE.EXE	SNA Base.
SNADMOD.DLL	Provides a network protocol-independent transport interface for SNA Server Client applications. On Windows 95 and Windows 98, this DLL allows applications written for the SNA Server Windows NT Client to run unmodified with the SNA Server Windows 95 Client. The SNADMOD.DLL functionality is provided by new DLLs on Windows 95 and Windows 98 (for example, SNAKRNL.DLL).

SNADUMP.DLL	SNA debug helper DLL.
SNAEVENT.DLL	SNA Server event log message file (used only on Windows NT).
SNAKRNL.DLL	Provides a network protocol-independent transport interface for SNA Server Client applications (used only on Windows 95 and Windows 98).
SNAPERF.DLL	Used for supporting performance monitoring (used only on Windows NT).
SNAPOPUP.EXE	Used for displaying SNA message pop-up dialog boxes (used only on Windows NT).
SNAREG.DLL	API support for accessing the registry.
SNASVC.DLL	Used for supporting services (used only on Windows NT).
SNATRACE.CPL	Control panel applet for dynamically enabling and disabling API (APPC, CPI-C, and LUA) tracing and event logging.
SNATRACE.HLP	Help file used for API (APPC, CPI-C, and LUA) tracing and event logging.
SNATRC.DLL	Used for SNA Server message tracing.
SNATRCCN.DLL	Used for SNA Server message tracing (used only on Windows NT).
SNATRCSN.DLL	Used for SNA Server message tracing.
WAPPC32.DLL	Used by 5250 emulators and other APPC applications.
WINAPPC.DLL	Stub APPC DLL used by 5250 emulators and other APPC applications.
WINCSV.DLL	Stub DLL used by emulators that need Common Service Verbs (CSV).
WINCSV32.DLL	Used by emulators that need CSV.
WINMGT32.DLL	Used by 5250 emulators and for supporting APPC management verbs.

SNA Server supports a number of different transport protocols. Depending on the transport protocol or protocols used by your customer, you will need to distribute some or all of the following files:

BVNSTT.DLL	Used to locate SNA Servers over Banyan VINES transport.
IBPCAST.DLL	Used to locate SNA Servers over TCP transport.
LMBCAST.DLL	Used to locate SNA Servers over Named Pipes transport.
NWSAP.DLL	Used to locate SNA Servers over NetWare transport.

SNABV.DLL	Banyan VINES transport for Windows NT.
SNACBV.DLL	Banyan VINES transport for Windows 95 and Windows 98.
SNACIP.DLL	Native TCP transport for Windows 95 and Windows 98.
SNACLM.DLL	Named Pipes transport for Windows 95 and Windows 98.
SNACNW.DLL	NetWare transport for Windows 95 and Windows 98.
SNAIP.DLL	Native TCP transport for Windows NT.
SNALM.DLL	Named Pipes transport for Windows NT.
SNANB.DLL	NetBIOS transport (used only on Windows NT).
SNANCP.DLL	Helper DLL for NetWare transport (required by SNACNW.DLL and SNANW.DLL).
SNANW.DLL	NetWare transport for Windows NT.
VSTAPI.DLL	Helper DLL for Banyan VINES transport (required by SNABV.DLL and SNACBV.DLL).

The following files are optional and are only necessary if your application uses these additional features supported by SNA Server:

AFTPAPI.DLL	Used for AFTP API support.
AFTPEVT.DLL	Used for AFTP API support.
AFTPMSG.DLL	Used for AFTP API support.
CHECKSUM.EXE	File checksum utility.
DLSSTAT.EXE	Distributed Link Service status utility for Windows NT.
LUASTR32.DLL	Translates LUA return codes into text strings.
SNANLS.DLL	Used for National Language Support (Windows 95 and Windows 98 only)
SNASHMEM.EXE	Tool to look at shared memory.
SNAVER.EXE	Tool to display version numbers.
TPSTART.EXE	Used for loading transaction programs (used on Windows NT only).
WCPIC32.DLL	Used for CPI-C API support.
WINCPIC.DLL	Used for CPI-C API support.
WINRUI.DLL	Stub DLL used for LUA API support.
WINRUI32.DLL	Used for LUA API support.
WINSLI.DLL	Stub DLL used for LUA API support.
WINSLI32.DLL	Used for LUA API support.

For SNA National Language Support using SNANLS, the appropriate NLS files will also need to be installed.

Debug files for the Windows NT DLLs are provided in the \CLIENTS\WINNT\SYSTEM\SYMBOLS\DLL directory, and debug files for the Windows 95 and Windows 98 DLLs are provided in the \CLIENTS\WIN95\SYSTEM\SYMBOLS\DLL directory.

The following files are necessary if your application needs to support Asian languages using the TrnsDT API or the SNANLS API:

TRNSDT.DLL	Used for Asian language support.
TRNSDTJ.DLL	Used for Asian language support.
TRNSDTK.DLL	Used for Asian language support.
TRNSDTS.DLL	Used for Asian language support.
TRNSDTT.DLL	Used for Asian language support.
SNADBC.TBL	Used for Asian language support.
SNADBCK.TBL	Used for Asian language support.
SNADBCS.TBL	Used for Asian language support.
SNADBCT.TBL	Used for Asian language support.
SNASBC.TBL	Used for Asian language support.
SNASBCK.TBL	Used for Asian language support.
SNASBCS.TBL	Used for Asian language support.
SNASBCT.TBL	Used for Asian language support.

The details of the installation process are described in the following topic.

The Installation Process in 32-bit Windows Environments

The ACME Setup tools used for installing SNA Server 4.0 and SNA Server 3.0 Client binaries are table-driven using a setup table file (STF), a text file containing installation information, and an SNAFILE.INF file. You may be using a different installation tool for your application.

Along with an installation and setup procedure, you should also supply an uninstall option to remove the SNA Server Client binaries and restore Windows 2000, Windows NT, Windows 98, or Windows 95 registry entries back to their original condition.

The general installation process for the Host Integration Server and SNA Server Client binaries is outlined below.

To install the Host Integration Server or SNA Server client binaries in 32-bit Windows environments

1. Verify that this system supports Host Integration Server or SNA Server. The installation tool should check for supported versions of the operating system and network operating system, adequate disk space, and any other requirements.
2. Copy files to target directories on the user's system. All of the Host Integration Server client binaries are normally installed in Host Integration Server\System directory on Windows 2000, Windows NT, Windows 98, and Windows 95 (typically C:\Program Files\Host Integration Server\System).
   For SNA Server 4.0, all of the SNA Server client binaries are normally installed in the SNA system directory. On Windows 2000 and Windows NT, the default locations is C:\SNA\SYSTEM. On Windows 98, and Windows 95, the default location is C:\SNA95\SYSTEM. Note that the SNA Server Client DLLs should be installed in the system directory of Windows 98 and Windows 95 (typically C:\WINDOWS\SYSTEM) on the user's system, not in the directory where the user has selected to install the SNA Server Client. The MFC, MSVC, and OLE run-time DLLs should be copied to an MFC40 subdirectory below this directory. This MFC40 subdirectory should be created if it does not exist. Copying all the DLLs to the Windows system directory on Windows 98 and Windows 95 and the SNA system directory on Windows 2000 and Windows NT makes it easy to ensure that there is only a single copy of a given DLL on the user's system.
3. The installation procedure must also check version resources for executable files and DLLs to avoid overwriting a newer copy already in the target directory with an older version from the distribution medium. All the Host Integration Server or SNA Server Client EXE files must be copied into the system directory to facilitate version checking when the user upgrades these files. The local binaries needed for your product would normally be copied to the subdirectory selected by the user during the installation process.
4. Define file locations and other parameters necessary to modify Windows 2000, Windows NT, Windows 98, or Windows 95, the network transport protocol, and other configuration files and registry entries. Based on these parameters, modify and update the registry and other required configuration files, saving copies of the original files.

Typical SNA Server Client Parameters in the Win32 Environment

The following table describes some typical SNA server client parameters for a Win32 environment.

Parameter	Common default
Root directory to which client files will be copied.	C:\SNA\SYSTEM on Windows NT and C:\SNA95\SYSTEM on Windows 95 and Windows 98
The path from which the SNA Server Client files are being installed.	CD-ROM drive
Extension to use when backing up machine's system files.	
Name of primary remote SNA Server.	
Name of backup remote SNA Server.	
The network transport protocol installed and used by the SNA Server Client on this machine.	
The version number of this machine's network operating system.	
For use with Novell NetWare, the user's choice of network subdomains for use with the SNA Server Client.	
For use with Banyan VINES, the user's choice of StreetTalk Group for use with the SNA Server Client.	

Registry Entries for Host Integration Server Administrator Client

This section describes the Microsoft Windows XP, Windows 2000, and Windows NT registry nodes and entries that need to be created or modified to support Host Integration Server 2000 Administrator Client binaries. Note that the Administrator Client runs only on Windows 2000 Professional or Windows NT 4.0 Workstation and will not run on Windows Millennium Edition, Windows 98, or Windows 95. Host Integration Server 2000 with Service Pack 1 add supports for running the Administrator client on Windows XP Professional.

If the Administrator Client is being used on Windows XP Professional, Windows 2000 Professional, or Windows NT Workstation, then the client configuration allows the SnaBase to run as an Application or a Service. If SnaBase is run as an Application (the default), then the registry settings match those documented for Clients Running the Host Integration Server 2000 End-User Client. If SnaBase is configured to run as a Service, then the registry settings that match those documented in this section on the Administrator Client apply.

This section contains:

Registry Settings for Administrator Client: SnaBase

The **SnaBase** registry node needs to be created if it does not already exist. It will contain configuration information used by the SNA Server Client binaries on Windows 2000 or Windows NT:

SYSTEM\CurrentControlSet\Services\SnaBase

Under this node, the following entries and values must be entered or modified:

DisplayName
The value of this entry should be set to an ASCIIZ string that will be displayed for the SNA Base. The value for this string should be set to SNABASE.

ImagePath
The value of this entry should be set to an ASCIIZ string that points to the SNABASE.EXE executable file. The typical value for this string is C:\SNA\SYSTEM\SNABASE.EXE.

Registry Settings for Administrator Client: SnaBase Parameters

The **Parameters** node needs to be created under the **SnaBase** node if it does not already exist. It will contain other parameters used by SNA Server Clients:

SYSTEM\CurrentControlSet\Services\SnaBase\Parameters

Under this node, the following entries and values must be entered or modified:

BufferAuditInterval
The value of this entry should be set to a DWORD that indicates the interval in seconds for buffer auditing. This registry entry is only applicable to Windows XP, Windows 2000, or Windows NT. A value of –1 indicates no buffer auditing. The value for this DWORD should be set to –1.

MaxRecordLength
The value of this entry should be set to a DWORD that indicates the maximum record length supported. The value for this DWORD should be set to 65535.

SNAServiceType
The value of this entry should be set to a DWORD that indicates the SNA Service type. The value for this DWORD should be set to 28 for SnaBase.

Transports
The value of this entry should be set to an REG_MULTI_SZ string that indicates the transport or transports that should be used for the SNA Server Client binaries. Possible values are:

SNABV.DLL (Banyan VINES)

SNAIP.DLL (TCP/IP)

SNALM.DLL (Microsoft Networking or Named Pipes)

SNANW.DLL (IPX/SPX)

The value for this string should be set based on the transport or transports to be used by the SNA Server Client.

Registry Settings for Administrator Client: SnaBase SnaTcp Parameters

The **SnaTcp** key and all parameters under it are optional.

If the TCP/IP transport is to be used, the SnaTcp node is created under the SnaBase\Parameters node if it does not already exist. This will contain other parameters used by SNA Server Clients:

SYSTEM\CurrentControlSet\Services\SnaBase\Parameters\SnaTcp

Under this node, the following entry and value are entered or modified:

NoDelay
The value of this entry should be set to an ASCIIZ string that indicates whether the TCP/IP transport should use the no delay option. Possible values are the "YES" or "NO" string. The value for this string should be set to the "YES" string.

Registry Settings for Administrator Client: SNA Server

The **SNA Server** registry node needs to be created if it does not already exist. It will contain configuration information used by the SNA Server Client binaries on Windows 2000 or Windows NT:

SOFTWARE\Microsoft\SNA Server

The CurrentVersion node needs to be created under the SNA Server node if it does not already exist. It will contain information used by SNA Server Client binaries on Windows NT:

SOFTWARE\Microsoft\SNA Server\CurrentVersion

The AdminAddons node needs to be created under the CurrentVersion node if it does not already exist. It will contain information used by SNA Server Client binaries on Windows NT:

SOFTWARE\Microsoft\SNA Server\CurrentVersion\AdminAddons

Under this node, the following entry and value must be entered or modified for supporting the SNA trace control panel applet:

snatrace
The value of this entry should be set to an ASCIIZ string that is the file name containing the SNA trace control panel applet. The value for this string should be set to SNATRACE.CPL.

Registry Entries for Host Integration Server End-User Client

This section describes the Win32 registry nodes and entries that need to be created or modified to support the Host Integration Server 2000 End-User Client binaries. Note that the End-User Client runs on Windows 2000 Professional, Windows NT 4.0 Workstation, Windows 98, and Windows 95. Host Integration Server 2000 with Service Pack 1 adds support for the following additional operating systems:

- Microsoft Windows XP Professional
- Microsoft Windows XP Home Edition
- Microsoft Windows Millennium Edition

The Host Integration Server Administrator Client will run only on Windows XP Professional, Windows 2000 Professional, or Windows NT 4.0 Workstation and cannot be run on Windows XP Home Edition, Windows Millennium Edition, Windows 98, or Windows 95. If the Administrator Client is being used on Windows XP Professional, Windows 2000 Professional, or Windows NT Workstation, then the client configuration allows the SnaBase to run as an Application or a Service. If SnaBase is run as an Application (the default), then the registry settings match those documented in the section on Clients Running Host Integration Server End-User Client. If SnaBase is configured to run as a Service, then the registry settings matching those documented in the section on the Administrator Client apply.

This section contains:

Registry Settings for End-User Client: SnaBase

The following registry node needs to be created if it does not already exist. It will contain configuration information used by the SNA Server Client binaries on Windows 98 and Windows 95:

SOFTWARE\Microsoft\SnaBase

Under this node, the following entries and values must be entered or modified:

DisplayName
The value of this entry should be set to an ASCIIZ string that will be displayed for the SNA Base. The value for this string should be set to SNABASE.

ImagePath
The value of this entry should be set to an ASCIIZ string that points to the SNABASE.EXE executable file. The typical value for this string is C:\SNA\SYSTEM\SNABASE.EXE.

Registry Settings for End-User Client: SnaBase Parameters

The **Parameters** node needs to be created under the **SnaBase** node if it does not already exist. It will contain other parameters used by SNA Server Clients:

SOFTWARE\Microsoft\SnaBase\Parameters

Under this node, the following entries and values must be entered or modified:

AutoTerminate
The value of this entry should be set to an ASCIIZ string that indicates whether autoterminate is enabled or disabled. This registry entry is only applicable to Windows 98 and Windows 95. Possible values are the "on" or "off" string. The value for this string should be set to the "off" string.

MaxRecordLength
The value of this entry should be set to a DWORD that indicates the maximum record length supported. The value for this DWORD should be set to 65535.

SNAServiceType
The value of this entry should be set to a DWORD that indicates the SNA Service type. The value for this DWORD should be set to 28 for SnaBase.

Transports
The value of this entry should be set to an ASCIIZ string that indicates the transport that should be used for the SNA Server Client binaries. Possible values are:

SNACBV.DLL (Banyan VINES)

SNACIP.DLL (TCP/IP)

SNACLM.DLL (Microsoft Networking or Named Pipes)

SNACNW.DLL (IPX/SPX)

The value for this string should be set based on the single transport to be used by the SNA Server Client.

Registry Settings for End-User Client: SnaBase Client Parameters

The **Client** node should be created under the **SnaBase\Parameters** node if it does not already exist. It will contain other parameters used by SNA Server Clients:

SOFTWARE\Microsoft\SnaBase\Parameters\Client

Under this node, the following entries and values must be entered or modified:

MaxRecordLength
The value of this entry should be set to a DWORD that indicates the maximum record length supported. The value for this DWORD should be set to 65535.

Registry Settings for End-User Client: SnaBase SnaTcp Parameters

The **SnaTcp** key and all parameters under it are optional.

If the TCP/IP transport is to be used, the SnaTcp node is created under the SnaBase\Parameters node if it does not already exist. This will contain other parameters used by SNA Server Clients:

SOFTWARE\Microsoft\SnaBase\Parameters\SnaTcp

Under this node, the following entry and value are entered or modified:

NoDelay
The value of this entry should be set to an ASCIIZ string that indicates whether the TCP/IP transport should use the no delay option. Possible values are the "YES" or "NO" string. The value for this string should be set to the "YES" string.

Registry Settings for End-User Client: SNA Server

The **SNA Server** registry node needs to be created if it does not already exist. It will contain configuration information used by the SNA Server Client binaries on Windows 98 and Windows 95:

SOFTWARE\Microsoft\SNA Server

The CurrentVersion node needs to be created under the SNA Server node if it does not already exist. It will contain information used by SNA Server Client binaries on Windows 98 and Windows 95:

SOFTWARE\Microsoft\SNA Server\CurrentVersion

Registry Settings for End-User Client: Windows Help

The following registry node needs to be created if it does not already exist. It will contain the names of the help files used by the SNA Server Client binaries on Windows 98 and Windows 95:

SOFTWARE\Microsoft\Windows\Help

Under this node, the following entries and values must be entered or modified for supporting the SNA Server Client help:

SnaBase.hlp
The value of this entry should be set to an ASCIIZ string that points to the directory where the SNABASE.HLP file is located. The typical value for this string is C:\SNA95\SYSTEM.

SnaTrace.hlp
The value of this entry should be set to an ASCIIZ string that points to the directory where the SNATRACE.HLP file is located. The typical value for this string is C:\SNA95\SYSTEM.

Client Setup for 16-bit Windows Environments

This section provides information regarding installation of the Microsoft® Host Integration Server and SNA Server Client binaries in the 16-bit Microsoft Windows® environment. This information will enable you to consolidate Host Integration Server and SNA Server into your product's own setup, simplifying operation for the end user.

SNA Server versions 3.0 and later support SNA Server versions 2.0 and 2.1 Client binaries for the 16-bit Windows environment, although it is recommended that you release updated binaries with each normal product release. This section describes the use of SNA Server versions 3.0 and later binaries, which are recommended for all new applications.

SNA Server Client Binary Files for 16-bit Windows Environments

Microsoft Host Integration Server 2000 client binary files for 16-bit Windows are located on the Host Integration Server CD-ROM in separate subdirectories under the Setup\Clients folder. The binary client files supplied with Host Integration Server for 16-bit Windows environments are as follows:

Subdirectory	Description
Win3x	Client binary files for use on Windows 3.x.
MS-DOS	Client binary files for use on MS-DOS.

The SNA Server client binary files for the 16-bit Windows environment are located in the \CLIENTS\WIN3x\SYSTEM directory on the SNA Server distribution CD-ROM. Additional copies are maintained on Library 2 on forum MSSNA on CompuServe. You will need at a minimum to distribute the following files:

File name	Purpose
APPCSTR.DLL	Translates APPC return codes into text strings.
CSVSTR.DLL	Translates Common Service Verb (CSV) return codes into text strings.
CTL3D.DLL	3-D controls (also distributed with other Windows applications).
FMISTR.DLL	Translates EIS/LUA errors into a text string.
SECURITY.DLL	Handles encryption and security
TOOLHELP.DLL	Toolhelp for debugging (also distributed with other Windows applications).
VER.DLL	Version numbers (also distributed with other Windows applications).
WDMOD.DLL	Provides a network protocol-independent transport interface for SNA Server Client applications.
WINAPPC.DLL	Used by 5250 emulators and other APPC applications.
WINCSV.DLL	Used by emulators that need Common Service Verbs.
WINMGT.DLL	Used by 5250 emulators.
WLOGTR.DLL	Used for API (APPC, CPI-C, and LUA) tracing and event logging.
WNAP.EXE	Keeps track of the names of SNA servers in the SNA Server (sub)domain.
WPOPUP.EXE	Message pop-up dialog boxes.

SNA Server supports a number of different transport protocols. Depending on the transport protocol or protocols used by your customer, you will need to distribute some or all of the following files:

BVCLI.DLL	Banyan VINES transport
IPCLI.DLL	Native TCP transport
LMCLI.DLL	Named Pipes transport
NBCLI.DLL	NetBIOS transport
NWCLI.DLL	NetWare transport

> SNA Server's use of IPX/SPX requires the Novell NetWare NWIPXSPX.DLL and NWNETAPI.DLL files, which are included on the SNA Server distribution CD-ROM. You must provide these files to your users under your own license agreement from Novell.

The following files are necessary if your application uses these additional features supported by SNA Server:

EHNAPPC.DLL	Used for enhanced APPC API support.
EHNRTW.DLL	Used for enhanced APPC API support.
LUASTR.DLL	Translates LUA return codes into text strings.
WINCPIC.DLL	Used for CPI-C API support.
WINRUI.DLL	Used for LUA API support.
WINSLI.DLL	Used for LUA API support.
WINTRC.DLL	Used for SNA Server message tracing.
YMGR.DLL	Used for DCA Comm Server 1.x support.

If you wish to support client tracing (which is a convenient troubleshooting tool), you will also need WINTRC.DLL and replacements for some of the files named above that can be found in the subdirectory \CLIENTS\WIN3x\TRACE.

The following files are necessary if your application needs to support Asian languages using the TrnsDT API:

TRNSDT.DLL	Used for Asian language support.
TRNSDTJ.DLL	Used for Asian language support.
TRNSDTK.DLL	Used for Asian language support.
TRNSDTS.DLL	Used for Asian language support.
TRNSDTT.DLL	Used for Asian language support.
SNADBC.TBL	Used for Asian language support.
SNADBCK.TBL	Used for Asian language support.
SNADBCS.TBL	Used for Asian language support.
SNADBCT.TBL	Used for Asian language support.
SNASBC.TBL	Used for Asian language support.
SNASBCK.TBL	Used for Asian language support.
SNASBCS.TBL	Used for Asian language support.
SNASBCT.TBL	Used for Asian language support.

The details of the installation process are described in the following topic.

The Installation Process in 16-bit Windows Environments

The ACME Setup tools used to install SNA Server 4.0 and 3.0 Client binaries are table-driven using an STF text file and an SNAFILE.INF. You may be using a different installation tool for your application.

Along with an installation and setup procedure, you should also supply an uninstall option to remove the SNA Server Client binaries and restore Microsoft MS-DOS® and Windows configuration files back to their original condition.

The general installation process for the SNA Server Client binaries is outlined below.

To install the SNA Server Client binaries in 16-bit Windows environments

1. Verify that this system supports SNA Server. The installation tool should check for supported versions of the operating system and network operating system, adequate disk space, and any other requirements.
2. Copy files to target directories on the user's system. Note that all the SNA Server Client DLLs must be installed in the Windows system directory on the user's system, not in the directory where the user has selected to install the SNA Server Client. Copying all the DLLs to the Windows system directory makes it simple to ensure that there is only a single copy of a given DLL on the user's system.
   The installation procedure must also check version resources for executable files and DLLs to avoid overwriting a newer copy already in the target directory with an older version from the distribution medium. All the EXE files must be copied into a directory specified in the [Wnap] WBinPATH location, to facilitate version checking when the user upgrades these files.
3. Define file locations and other parameters necessary to modify MS-DOS, the network operating system, and other configuration files. Based on these parameters, modify and update AUTOEXEC.BAT, LANMAN.INI, and other required configuration files, saving copies of the original files.
4. Update the WIN.INI file, creating a [Wnap] section. Add the necessary entries within this section, saving a copy of the original file.
5. If necessary, modify SYSTEM.INI and save a copy of the original file.

Typical SNA Server Client Parameters in 16-bit Windows Environments

The following table describes some typical SNA Server client parameters in a 16-bit environment.

Parameter	Common default
Root directory to which client files will be copied.	C:\SNA.WIN
The path from which the SNA Server Client files are being installed.	CD-ROM drive
Extension to use when backing up machine's system files.	
Name of primary remote SNA Server.	
Name of backup remote SNA Server.	
The network operating system installed on this machine.	
The version number of this machine's network operating system.	
For use with Novell NetWare, the user's choice of network domains for use with the SNA Server Client.	
For use with Banyan VINES, the user's choice of StreetTalk Group for use with the SNA Server Client.	

Modifications to WIN.INI for SNA Server Clients

This section describes the [Wnap] section that must be created in the WIN.INI file for the SNA Server Client installation.

[Wnap]

NetSetup =

NosSetup =

NosType =

Remote =

SecureSponsor =

StGroupName =

WBinPath =

Entries

NetSetup
This flag indicates whether this was a network setup of SNA Server Client. Valid values are YES or NO.

NosSetup
This string specifies the network operating system for which the SNA Server Client was set up. The **NosSetup** entry is used only by the setup program; the other SNA programs ignore it.

Valid values for NosSetup are:

LANMAN

NOVELL

TCPIP

VINES

NosType
This string specifies the transport DLL that the WNAP/WDMOD will use when communicating with the SNA server.

Valid values for NosType are:

LANMAN

NETBIOS

NOVELL

TCPIP

VINES

Remote
This is a required keyword in WIN.INI. This string specifies the location of the sponsoring SNA server. The SNA Server Client uses **Remote** to find a sponsoring SNA server. After the client establishes a "sponsor" connection with one of these SNA servers, it can discover all available SNA resources in the subdomain of which the sponsoring server is a member.

Valid values for Remote depend on the network operating system:

- For BANYAN, **Remote** specifies a subdomain name. The client searches in the StreetTalk group specified in **STGroupName** for a sponsoring SNA server configured to use the specified subdomain name.
- For LANMAN and WFW, Remote specifies zero to two server names. If no servers are given, for example, by entering "Remote=", the client looks for a sponsoring SNA server in its own domain. The server names must be of the form \\<nodeName>, and if a backup is specified, it must be separated from the primary by a blank. The Remote entry can also specify a subdomain name. If the value does not start with two backslashes, it is treated as a subdomain name.
- For NETWARE, Remote specifies the subdomain name specified at the SNA server when it is configured to use IPX/SPX transport.
- For TCP, Remote specifies node names, which can be specified as \\<nodeName>, or in Domain Name System (DNS) syntax (<node>.<net>.<org>), or by IP address (<ddd>.<ddd>.<ddd>.<ddd>). The Remote entry can also specify a

subdomain name. If the value does not start with two backslashes, it is treated as a subdomain name.

SecureSponsor

This string indicates whether the sponsor connection will be secured with data encryption. Valid values are YES or NO. The default value for this parameter is NO.

StGroupName

This string specifies the StreetTalk group name if using VINES as the network operating system.

WBinPath

This string specifies the location where the client SNA files on the local machine were installed. The default value for this path is C:\SNA.WIN.

Modifications to SYSTEM.INI for SNA Server Clients

This section describes the modifications that must be made to the SYSTEM.INI file for the SNA Server Client installation.

[386Enh]

NetHeapSize =

Entries

NetHeapSize

This number specifies the maximum buffer size (in kilobytes) that Windows enhanced mode allocates in conventional memory for data transfers. The default value is 12, and all values are rounded up to the nearest 4K. For SNA Server Clients this value must be set to 32 or greater. Some network operating systems may require an even larger value.

Appendices and Glossary

This section contains appendices and a glossary for programmers developing applications by using Microsoft® Host Integration Server 2000.

This section contains:

Common Abbreviations

This section of the Microsoft Host Integration Server 2000 Developer's Guide provides information on the abbreviations used in Microsoft® Host Integration Server 2000 SDK documentation.

Abbreviation	Description
ACKRQD	acknowledgment required
API	application programming interface
APPC	Advanced Program-to-Program Communications
ASCII	American Standard Code for Information Interchange
BBI	begin bracket indicator
BBIUI	begin basic information unit indicator
BCI	begin chain indicator
BETB	between-brackets
BICB	bind information control block
CCITT	Comité Consultatif Internationale de Télégraphie et Téléphonie
CDI	change direction indicator
CEI	chain ending indicator
CICB	connection information control block
CPI-C	Common Programming Interface for Communications
CRC	cyclical redundancy check
CSV	common service verb
CTS	clear to send
DAF	destination address field
DFC	data flow control
DFT	distributed function terminal
DLC	data link control
DLL	dynamic-link library
DMA	direct memory access
DMOD	dynamic access module
DR1I	definite response 1 indicator
DR2I	definite response 2 indicator
DTE	data terminal equipment
DTR	data terminal ready
EBCDIC	Extended Binary Coded Decimal Interchange Code
EBI	end bracket indicator
EBIUI	end basic information unit indicator
ECI	end chain indicator
EDI	enciphered data indicator
ERI	exception response indicator
EXR	exception request
FMD	function management data
FMHI	function management header indicator
FMI	Function Management Interface
GDS	general data stream
HDLC	High-Level Data Link Control
IHV	independent hardware vendor
INB	in-brackets
IRP	I/O request packet
K	kilobyte
LAN	local area network
LL	record length
LPI	locality, partner, index
LU	logical unit

LUA	conventional Logical Unit Application programming interface
MAC	media access control
MB	megabyte
MDI	multiple document interface
NAP	Network Access Program
NAU	network addressable unit
NC	network control
NCP	Network Control Panel
NSPE	network services procedure error
NMVT	Network Management Vector Transport
OAF	originating address field
OEM	original equipment manufacturer
OS	operating system
PC	personal computer
PDI	padded data indicator
PIP	program initialization parameter
PLU	primary logical unit
PU	physical unit
PVC	permanent virtual circuit
QLLC	qualified logical link control
QRI	queued response indicator
RH	request/response header
RNR	receive not ready
RTI	response-type-indicator
RTM	Response Time Monitor
RTS	request to send
RU	request/response unit
RUI	Request Unit Interface
SAA	Systems Application Architecture
SAP	service access point
SC	session control
SCM	Service Control Manager
SDI	sense-data-included indicator (in LUA) system detected error indicator (in EIS)
SDLC	Synchronous Data Link Control
SLI	Session Level Interface
SNA	Systems Network Architecture
SNADIS	SNA Device Interface Specification
SNRM	set normal response mode
SRL	Setup Resource Library
SSCP	system services control point
STSN	set and test sequence number
SVC	switched virtual circuit
TH	transmission header
TP	transaction program
TS	transmission service
TSR	terminate-and-stay-resident
UA	unnumbered acknowledgement
VCB	verb control block
VTAM	Virtual Telecommunications Access Method
XID	exchange identification

Glossary

This section contains the glossary for programmers developing applications using Microsoft Host Integration Server 2000.

This glossary includes terms and definitions from:

The American National Standard Dictionary for Information Systems:

- *ANSI X3.172-1990*, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- *The ANSI/EIA Standard--440-A, Fiber Optic Terminology*. Copies may be purchased from the Electronic Industries Association, 2001 Pennsylvania Avenue, N.W., Washington, DC 20006. Definitions are identified by the symbol (E) after the definition.
- The Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.
- The Network Working Group Request for Comments: 1208.
- The IBM Dictionary of Computing, New York: McGraw-Hill, 1994.
- The Object-Oriented Interface Design: IBM Common User Access Guidelines, Carmel, Indiana: Que, 1992.›

The following cross-references are used in this glossary:

Synonym for: Indicates that the term has the same meaning as a preferred term defined in the glossary.

Synonymous with: References all other terms that have the same meaning.

See also: Refers the reader to terms that have a related, but not synonymous, meaning.

This section contains:

Glossary

3270
The information display system for IBM hosts (mainframes). The system includes terminals, printers, and controllers that allow a user to access host functions.

5250 emulator
Software that allows a PC to act as a 5250 terminal interacting with an AS/400 system.

A

A3270
The server transaction program for the APPC 3270 Terminal Emulator facility.

ACTLU
SNA command sent by the SSCP to an LU to activate a session and establish session parameters.

ACTPU
SNA command sent by the SSCP to activate a PU, so that any logical units controlled by this PU are available to the SNA network.

Advanced Peer-To-Peer Networking (APPN)
An extension to SNA that features (a) greater distributed network control that avoids critical hierarchical dependencies, thereby isolating the effects of single points of failure; (b) dynamic exchange of network topology information to foster ease of connection, reconfiguration, and adaptive route selection; (c) dynamic definition of network resources; and (d) automated resource registration and directory lookup. APPN extends the LU 6.2 peer orientation for end-user services to network control and supports multiple LU types, including LU 2, LU 3, and LU 6.2.

Advanced Program-to-Program Communication (APPC)
(1) The general term that characterizes the LU 6.2 architecture and its various implementations in products. (2) Sometimes used to refer to the LU 6.2 architecture and its product implementations as a whole, or to an LU 6.2 product feature in particular, such as an APPC application programming interface. (3) A method for allowing programs to communicate directly with each other across a network or within a single system. APPC uses a type of LU called LU 6.2, and allows TPs to engage in peer-to-peer communications in an SNA environment.

AFTP
(1) APPC File Transfer Protocol. (2) The client transaction program for the APPC File Transfer Protocol facility. (3) An interactive full-screen environment with a specific set of commands used to manage and transfer files between a client and server computer. (4) An API that provides APPC file transfer capabilities.

AFTPD
The server transaction program for the APPC File Transfer Protocol facility.

alert
A message sent to indicate an abnormal event or a failure.

allocate
(1) The process an operating system uses to respond to a request from a program to reserve memory for use by the program. (2) In APPC, a verb that assigns a session to a conversation. *Contrast with* deallocate.

APING
(1) The APPC Connectivity Tester facility. (2) The client transaction program for the APPC Connectivity Tester facility.

APINGD
The server transaction program for the APPC Connectivity Tester facility.

APPC verb
The mechanism by which a program accesses APPC. Each verb supplies parameters to APPC.

application programming interface (API)
The set of programming language constructs or statements that can be coded in an application program to invoke the specific functions and services provided by an underlying operating system or service program.

application TP
An application program that uses APPC to accomplish tasks for users and exchange data with other TPs in an SNA environment.

ASCII
American Standard Code for Information Interchange. A coding scheme that assigns numeric values to letters, numbers, punctuation marks, and certain other characters.

asynchronous verb completion
Processing of an SNA verb where the initial API call returns immediately, so that the normal operation of the program is not blocked while processing completes. When the verb completes, the application is notified through a Windows message or event. *Contrast with* synchronous verb completion.

B

Base
A part of each Host Integration Server 2000 component that provides the operating environment for the core functions of that component. The Base passes messages between components and provides functions common to all components, such as diagnostic tracing.
basic conversation
In APPC, a conversation type generally used by programs that provide services to other local programs. Basic conversations provide a greater degee of control over the transmission and handling of data than mapped conversations. *See also* mapped conversation.
blocking
A method of operation in which a program that issues a call does not regain control until the call completes. *See also* synchronous verb completion.
bracket
A chained set of RUs and their responses, which together make up a transaction between two LUs. One bracket must be finished before another can be started.

C

characteristics
A set of internal values maintained by CPI-C for each conversation. They can affect the operation of the entire conversation or of specific calls.
client
(1) A functional unit that receives shared services from a server. (2) A user.
code page
A table that associates specific ASCII or EBCDIC values with specific characters.
Common Programming Interface for Communications (CPI-C)
An evolving application programming interface (API), embracing functions to meet the growing demands from different application environments and to achieve openness as an industry standard for communications programming. CPI-C provides access to interprogram services such as (a) sending and receiving data, (b) synchronizing processing between programs, and (c) notifying a partner of errors in the communication.
Common Service Verb (CSV)
An API that provides ways of translating characters, tracing, and sending network management information to a host.
configuration file
A file containing setup and configuration information for Host Integration Server 2000. It defines servers, connections, LUs, users, and other items. The configuration file that is loaded when Host Integration Server 2000 Manager starts is called COM.CFG.
connection object
In AFTP, a connection (not necessarily active) to a partner computer.
connectivity
(1) The capability of a system or device to be attached to other systems or devices without modification. (2) The capability to attach a variety of functional units without modifying them.
conversation
(1) A logical connection between two transaction programs using an LU 6.2 session. Conversations are delimited by brackets to gain exclusive use of a session. (2) The interaction between TPs carrying out a specific task. Each conversation requires an LU-LU session. A TP can be involved in several conversations simultaneously. *See also* basic conversation, mapped conversation. (CPI-C definition) The logical connection between two programs that allows the programs to communicate with each other.
CSV verb
The mechanism by which a program accesses CSV. Each verb supplies parameters to CSV. *See also* Common Service Verb.
current directory
The first directory in which the operating system looks for programs and data files and stores files for output.

D

database
(1) A collection of data with a given structure for accepting, storing, and providing on demand, data for multiple users. (2) A collection of interrelated data organized according to a database schema to serve one or more applications. (3) A collection of data fundamental to a system. (4) A collection of data fundamental to an enterprise.
data link control (DLC)
In SNA, the protocol stack layer that transmits messages across links and manages link-level flow and error recovery.
data set members
Members of partitioned data sets that are individually named elements of a larger file that can be retrieved by name.
deallocate
(1) The process an operating system uses to free memory previously allocated by a program. (2) In APPC, a verb that ends a

conversation. *Contrast with* allocate.

distributed function terminal (DFT)
A type of intelligent terminal supported by IBM 3270 control units, in which some of the terminal's functions are controlled by the terminal and others by the control unit. It allows multiple sessions and connects to host systems or to peer systems through host systems. DFTs are often connected using coaxial cable.

directory
(1) A list of files that are stored on a disk or diskette. A directory also contains information about the files such as size and date of last change. (2) A named grouping of files in a file system.

display verb
An APPC verb that returns configuration information and current operating values for a computer running Host Integration Server 2000.

DL-BASE
The type of Base used by Host Integration Server 2000 3270 emulation programs. It supports a single Host Integration Server 2000 component or a single user application and has entry points for initialization, sending messages, receiving messages, and termination. *See also* Base.

Dynamic Access Module (DMOD)
An SNA component that provides the communications facilities needed to pass messages between the Bases.

dynamic locality
A locality in a Host Integration Server 2000 system where the localities are not configured in advance. In this type of system the relationships between localities are configured dynamically as localities enter and leave the system. *See also* locality.

E

EBCDIC
Extended Binary Coded Decimal Interchange Code. A coding scheme developed by IBM for use with its computers as a standard method of assigning binary (numeric) values to alphabetic, numeric, punctuation, and transmission-control characters.

F

fill type
A value used in basic conversations that indicates whether programs receive data in the form of logical records or as a specified length of data.

flow
A verb flows from one LU to another.

Format 0 XID
A type of XID that supplies minimal information about a node. These XIDs have a fixed length. They can be used for 3270 and LUA communications, but not for APPC. *See also* Format 3 XID *and* XID.

Format 3 XID
A type of XID that supplies more detailed information about a node than a Format 0 XID. These XIDs have a variable length. They can be used for APPC as well as for 3270 and LUA communication. *See also* Format 0 XID *and* XID.

full-duplex transmission
Two-way electronic communication that takes place in both directions simultaneously. *Contrast with* half-duplex transmission.

fully qualified LU name
The two-part network address (network.lu) that uniquely identifies a destination (typically a user) in the network.

Function Management Interface (FMI)
An interface that provides applications with direct access to SNA data flow and information about SNA control flows by means of status messages. It is particularly suited to the requirements of 3270 emulation applications.

H

half-duplex transmission
Two-way electronic communication that takes place in only one direction at a time. *Contrast with* full-duplex transmission.

HLLAPI
High-Level Language Application Programming Interface. An API that allows you to develop and run programmer-operator applications on IBM PCs (or compatibles) that communicate with IBM mainframes using 3270 emulation.

Host Integration Server
A Microsoft software program that allows a PC to communicate with remote computers such as IBM mainframes, AS/400s, or other PCs on an SNA network.

hot backup
(1) The ability to take systems online and offline without disrupting service. (2) A configuration in which one resource (such as a server running Host Integration Server 2000 software) can automatically handle sessions if another cannot. Such servers can provide hot backup for 3270, LUA, or downstream sessions through pools containing LUs from multiple servers. Servers

running Host Integration Server 2000 software can provide hot backup for 5250 terminal emulation through the use of LU names that are the same on multiple servers.

I

invokable
Capability of a program to be started by another program. For example, an invokable APPC TP can be started in response to a request from another TP.

invokable TP
A TP that can be invoked by another TP.

invoked program
A program that has been activated by a call or verb. *See also* invoking program.

invoked TP
A TP started by another (invoking) TP. The invoked TP is started by an allocate verb sent to it by the invoking TP.

invoking program
A program that uses a call or verb to activate another program. *See also* invoked program.

invoking TP
A TP that initiates a conversation with another TP. The invoking TP starts the other TP by instructing the remote node to load the invokable TP.

L

link service
The software component of Host Integration Server 2000 that communicates with the device driver for a particular communication adapter (802.2, SDLC, X.25, DFT, Channel, or Twinax).

local LU
In an APPC or CPI-C conversation, the LU on the local end. *Contrast with* partner LU *and* remote LU.

local program
In CPI-C, the program on the local end of the conversation. *Contrast with* partner program.

local TP
In an APPC or CPI-C conversation, the TP on the local end. *Contrast with* partner TP *and* remote TP.

locality
A Base and the components within it; that is, a Host Integration Server 2000 executable program. *See also* dynamic locality.

logical unit (LU)
(1) A type of network-accessible unit that enables users to gain access to network resources and communicate with each other. (2) A preset unit containing all the configuration information needed for a user, program, or downstream system to establish a session with a host or peer computer. Host Integration Server 2000 offers four types of LUs: 3270 LUs are used for 3270 terminal emulation; LU type 6.2 LUs are used for APPC; LUA LUs are used with programs written for the LUA interface; downstream LUs are used to allow downstream systems to connect to a host through a computer running Host Integration Server 2000. On a computer running Host Integration Server 2000, LUs are assigned to connections, and the connections and LUs work together to provide access to other systems on the SNA network.

Logical Unit Application (LUA)
A conventional LU application, or the interface that these applications use. LUA allows workstations to communicate with host applications using LU 0, 1, 2, or 3 protocols.

LPI address
Used to identify each end of a connection between two partners. It can have three components: L identifies the locality, P identifies the partner within the locality, and I identifies a logical entity within the partner. *See also* locality *and* partner.

LU 6.2
A protocol used by two TPs communicating as peers. LU 6.2 works in combination with node type 2.1 to provide APPC communications using independent LUs. LU 6.2 also works with node type 2.0 to provide APPC communications with dependent LUs.

LU alias
The string that identifies an LU to a TP. It can be up to eight characters long.

LU pool
A number of LUs, all of the same kind, that are made available as a group. A user, LUA application, or downstream system using the pool can get LU access as long as one of the pooled LUs is available. This provides for better access for multiple users, applications, or downstream systems in an SNA installation.

LU-LU session
The communication between two LUs over a specific connection for a specific amount of time. An LU-LU session is needed for two TPs to interact. One session can be used serially by many TPs. *See also* multiple sessions *and* parallel sessions.

M

MAC address
A 12-byte hexadecimal address used by the media access control layer of an 802.2 connection. It corresponds to the VTAM MACADDR= parameter and to the remote network access parameter for an 802.2 connection with Host Integration Server 2000.

mapped conversation
A conversation type generally used by programs that accomplish tasks for users. In a mapped conversation, the sending program sends one record at a time, and the receiving program receives one record at a time. The characters **MC_** at the beginning of a verb stand for mapped conversation. *See also* basic conversation.

mode name
The name used by the initiator of a session to designate the characteristics desired for the session, such as traffic pacing values, message-length limits, Sync Point and cryptography options, and the class of service within the transport network.

multiple sessions
In CPI-C, two or more concurrent sessions with different partner LUs. *See* LU-LU session.

MVS
Multiple Virtual Storage. Implies MVS/370, the MVS/XA product, and the MVS/ESA product.

N

node
A server, controller, workstation, printer, or other processor that implements SNA functions.

SNA defines three kinds of nodes: the host subarea node that controls and manages a network; the communication controller subarea node that routes and controls data flow through the network; and peripheral nodes that include printers, workstations, cluster controllers, and distributed processors.

node type 2.1
An SNA component, such as an intelligent terminal or a PC, that works together with LU type 6.2 to support peer-to-peer communications, allowing the LUs to function independently from the host. This is referred to as a 2.1 node.

NT operating system
Microsoft Windows NT Server operating system.

P

packet
In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. The data, control signals, and, possibly, error control information are arranged in a specific format.

parallel sessions
In CPI-C, two or more concurrent sessions with the same partner LU. *See* LU-LU session.

partitioned data set (PDS)
A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

partner
An addressable component of a locality; that is, code to which messages can be sent. *See also* locality.

partner LU
In an APPC or CPI-C conversation, the LU on the far end. The partner LU serves the partner TP. *Contrast with* local LU. *See also* partner TP *and* remote LU.

partner program
For CPI-C, the program receiving the CPI-C call.

partner TP
In an APPC or CPI-C conversation, the TP on the far end. *Contrast with* local TP. *See also* partner LU *and* remote TP.

password
A string of characters that a user, a program, or a computer operator must specify to meet security requirements before gaining access to a system and to the information stored within it.

path
A path exists between two localities when the DMODs in the localities can successfully pass messages between them. A path must exist between two localities before a connection can exist between partners in these localities. *See also* DMOD *and* locality.

pattern-matching character
A special character such as an asterisk (*) or a question mark (?) that can be used to represent one or more characters. Any character or set of characters can replace a pattern-matching character. *Synonymous with* wildcard character.

peer-to-peer
A type of communication in which two systems communicate as equal partners sharing the processing and control of the exchange, as opposed to host-terminal communication in which the host does most of the processing and controls the exchange.

physical unit (PU)
A network-addressable unit that provides the services needed to use and manage a particular device, such as a communications link device. A PU is implemented with a combination of hardware, software, and microcode.
pipe
A portion of memory that can be used by one process to pass information along to another.
protocol
(1) A set of semantic and syntactic rules that determine the behavior of functional units in achieving communication. (2) In Open Systems Interconnection architecture, a set of semantic and syntactic rules that determine the behavior of entities in the same layer in performing communication functions. (3) In SNA, the meanings of, and the sequencing rules for, requests and responses used for managing the network, transferring data, and synchronizing the states of network components.
PU 2.0
In an SNA network, the component that defines controller and terminal-type resources similar to an IBM 3274 Control Unit.
PU 2.1
In an SNA network, a component such as an intelligent terminal or a PC that works together with LU type 6.2 to support peer-to-peer communications, allowing LUs to function independently from the host.

Q

QLLC
Qualified logical link control. The protocol that permits SNA sessions to occur over X.25 networks.
queued TP
An invokable TP that can be started by only one incoming allocate command at a time. Incoming allocate commands that arrive while the queued TP is running do not start the program again, but are queued until the program issues another **RECEIVE_ALLOCATE** or until it finishes execution.

R

race condition
A condition in which a feedback circuit interacts with the internal circuit processes in a way that produces chaotic output behavior.
remote LU
In an APPC or CPI-C conversation, the LU on the remote end. *Contrast with* local LU. *See also* remote TP.
remote TP
In an APPC or CPI-C conversation, the TP on the remote end. *Contrast with* local TP. *See also* remote LU.
Request Unit Interface (RUI)
A basic interface that allows programs to acquire and release control of conventional LUs. The RUI also reads and writes request/response headers (RHs), transmission headers (THs), and request unit (RU) data. *Contrast with* Session Level Interface.
root
The topmost node in a directory structure.
root directory
The first directory on a drive in which all other files and subdirectories exist.

S

semaphore
A flag variable that is used to govern access to shared system resources.
server
(1) A functional unit that provides shared services to workstations over a network; for example, a file server, a print server, a mail server. (2) In a network, a data station that provides facilities to other stations; for example, a file server, a print server, a mail server.
service TP
A TP that uses basic conversation verbs to provide services to other TPs.
session
(1) In network architecture, for the purpose of data communication between functional units, all the activities that take place during the establishment, maintenance, and release of the connection. (2) A logical connection between two network-accessible units (NAUs) that can be activated, tailored to provide various protocols, and deactivated, as requested. Each session is uniquely identified in a transmission header (TH) accompanying any transmissions exchanged during the session.
Session Level Interface (SLI)
A higher-level interface that facilitates the opening and closing of SNA sessions with host LU 0, LU 1, LU 2, and LU 3 application programs. The SLI permits application programs to control the data traffic at a logical message level. *Contrast with* Request Unit Interface.
side information table

In CPI-C, a table that stores the initialization information required for two programs to communicate The table resides in the operating system's memory and the system administrator maintains it by accessing a symbolic destination name. The table is derived from the configuration file for Host Integration Server 2000.

SnaBase
The SNA Workstation Process. It is present at all times on PCs that wish to participate in the SNA network and on PCs where dynamic loading is to be performed.

SNALink
Link support software that integrates hardware components into a Host Integration Server 2000 system. An SNALink is defined when a Host Integration Server 2000 system is installed. An SNALink can support only one physical connection from the server.

Host Integration Server 2000
A Microsoft software program that allows a PC to communicate with remote computers such as IBM mainframes, AS/400s, or other PCs on an SNA network.

subdirectory
A directory contained within another directory in a file system hierarchy.

switched virtual circuit (SVC)
A type of circuit used by an X.25 connection, where the circuit is not constantly active, but is called and cleared dynamically., The destination address is supplied when the circuit is called.

synchronous data link control (SDLC)
A type of link service used for managing synchronous data transfer over standard telephone lines (switched lines) or leased lines.

synchronous verb completion
Processing of an SNA verb where the operation of the program is blocked until processing completes. *Contrast with* asynchronous verb completion.

system services control point (SSCP)
(1) A host-system network component that provides network services for dependent nodes. (2) An SNA network component that helps control and maintain communication flow between PUs and LUs on the network. Multiple SSCPs can work together to coordinate communications.

Systems Network Architecture (SNA)
The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks. The layered structure of SNA allows the ultimate origins and destinations of information, that is, the end users, to be independent of and unaffected by the specific SNA network services and facilities used for information exchange.

A collection of rules that brings uniformity to communications systems and the ways they interact. These rules define various functions that allow information to be transferred from one computer to another in a form that is usable by the receiving computer.

T

transaction
A processing task accomplished by programs using APPC or CPI-C.

transaction program (TP)
(1) An application program that uses APPC or CPI-C to exchange data with another TP on a peer-to-peer basis. (2) A program that processes transactions in an SNA network. There are two kinds of transaction programs: application transaction programs and service transaction programs. *See also* conversation.

U

user identifier
A string of characters that uniquely identifies a user to a system.

V

verb
Command from one LU to another to exchange data and perform tasks. *See also* APPC verb.

verb control block (VCB)
A structure made up of variables, which identifies the verb to be executed, supplies information to be used by the verb, and contains information returned by the verb when execution is complete.

VM
Virtual machine.

W

wildcard character
Synonym for pattern-matching character.

X

X.25
The CCITT standard used for communication over a packet-switching network. X.25 uses the QLLC (qualified logical link control) protocol.

XID
Exchange Identification. An identifier that is exchanged between nodes on an SNA network, and that allows the nodes to recognize each other and to establish link and node characteristics for communicating. With Host Integration Server 2000, two kinds of XIDs can be exchanged: Format 0 XIDs (containing only basic information) and Format 3 XIDs (containing more detailed information). *See also* Format 0 XID *and* Format 3 XID.

Accessing Host Transactions in .NET-based Applications Using COMTI

Microsoft Corporation

July 2003

Applies to:
Microsoft® Host Integration Server 2000

Summary: Microsoft Host Integration Server 2000 includes a sample application called CedarBank. The CedarBank sample is a simple banking application made up of a collection of COMTI (COM Transaction Integrator) type libraries, COBOL samples, and client applications. It was created to demonstrate the various ways of using COMTI to access CICS (Customer Information Control System) or IMS (Information Management System) transactions running on an IBM host.

Since many customers are using Microsoft Visual Studio® .NET and the .NET Framework for new development work and need the ability to use COMTI in their projects, this document describes some .NET-based versions of the existing CedarBank samples provided with Host Integration Server 2000, and points out the key concepts to keep in mind when using COMTI with the .NET Framework.

Download the COMTI technical article and its accompanying sample.

Administration and Management of Data Access Using the OLE DB Provider for DB2

Host Integration Server 2000
Microsoft Corporation

October 2001

Summary: Microsoft Host Integration Server 2000 (HIS) includes a number of components that enable integration with host data sources. This article describes features and tools for administering and managing data access to IBM DB2 relational database systems using the OLE DB Provider for DB2 supplied with Microsoft Host Integration Server 2000. (33 printed pages)

Contents

Introduction

Microsoft® Host Integration Server 2000 includes a rich set of *Data Integration* components, which provide desktop or server-based applications with direct access to host data. These *Data Integration* components provide a comprehensive set of data access services, which includes direct data access to relational and non-relational mainframe and AS/400 data through open database connectivity (ODBC), object linking and embedding database (OLE DB), and COM automation controls.

The *Data Integration* components included in Host Integration Server 2000 provide access to both structured and non-structured data stored on IBM mainframe or AS/400 computers. This data can be stored in a database or file system. In addition to data access, the *Data Integration* components also provide data transfer services between Microsoft® Windows® 2000 computers and host systems.

The *Data Integration* components can be organized into the following categories:

- Relational database access
- Record file access
- File transfer
- AS/400 data queue access

All of these services make use of IBM host-based products that implement the IBM Distributed Data Management Architecture (DDM). DDM is a framework or methodology for sharing and accessing data between systems. DDM defines the "how to communicate" and leaves it up to individual platform vendors to implement the DDM architecture. IBM currently supports DDM for most IBM platforms, including: OS/390 (MVS), AS/400, RS/6000 (AIX), and AS/36.

Much of the operational data stored on OS/390, AS/400, and RS/6000 computers is accessed via a relational database management system. The most popular database on these host systems is IBM DB2. Host Integration Server 2000 offers relational database access by using the Distributed Relational Data Architecture (DRDA) subset of DDM.

DRDA offers both Remote Unit of Work (RUW) and Distributed Unit of Work (DUW) access to host data. RUW is used for read-only and simple updating of database tables using SQL statements and stored procedures. DUW is used when updates span multiple DB2 instances or computer systems and supports the two-phase commit (2PC) protocol. The 2PC protocol ensures that changes to multiple databases will either succeed or fail in their entirety.

Host Integration Server 2000 implements access to DB2 via two features:

- Microsoft OLE DB Provider for DB2
- Microsoft ODBC Driver for DB2

The Microsoft OLE DB Provider for DB2 relies on an underlying DRDA application requester (AR) developed by Microsoft. The DRDA AR connects the OLE DB Provider for DB2 to DB2 on popular platforms, including OS/390, OS/400, RS/6000-AIX, Microsoft® Windows NT®, and Windows 2000.

The OLE DB Provider for DB2 supports a number of data access features including static and dynamic SQL, execution of DB2 stored procedures, transactions using two phase commit, and network connectivity using SNA LU 6.2 or TCP/IP. Developers can use Visual C or Microsoft Visual C++® to integrate DB2 data with Web-based and Windows-based applications. Microsoft Visual Basic® and Web developers (using scripting languages such VBScript) can use the higher-level Microsoft ActiveX® Data Objects (ADO) to develop e-commerce solutions. Additionally, DB2 is directly accessible from productivity applications, such as Microsoft Office 2000 using Visual Basic for Applications (VBA) and ADO from within Microsoft Excel.

This article describes features and tools for administering and managing data access to IBM DB2 relational database systems using the OLE DB Provider for DB2. Companion articles will discuss administration and management of other data integration components provided in Host Integration Server 2000.

Configuring and Managing Data Sources

Microsoft® Data Access Components 2.0 and later includes Data Links, a generic method for managing and loading connections to OLE DB data sources. Microsoft Data Links, a core element of the Microsoft Data Access Components (MDAC), provide a uniform method of creating persistent OLE DB data source object definitions stored in the form of universal data link (UDL) files. The OLE DB Provider for DB2 normally uses Data Links and UDL files for loading and configuring data sources.

Applications, such as the RowsetViewer sample from the Microsoft Data Access SDK, can open created UDL files and pass the stored initialization string to the OLE DB Provider for DB2 at run time. Data Links provide a flexible method for finding and saving connection information to OLE DB data sources.

In order to use Microsoft OLE DB Provider for DB2 with an OLE DB consumer application, the user must either (1) create a Microsoft data link (UDL) file and call this from the application, or (2) call the OLE DB provider from within the application using a connection string that includes the provider name and any other needed parameters.

Configuring Data Sources for the OLE DB Provider for DB2

Data source information must be configured for each DB2 system data source object that is to be accessed using the OLE DB Provider for DB2. The default parameters for the OLE DB Provider for DB2 are used as the default values for data sources and when these parameters are not configured for each data source.

Microsoft Data Links provides a uniform method for creating file-persistent OLE DB data source object definitions in the form of Universal Data Link (UDL) files. Applications, such as the RowsetViewer sample included in Microsoft Data Access and the Platform SDK, can open created UDL files and pass the stored initialization string to the OLE DB Provider for DB2 at run time.

Creating New Data Links for the OLE DB Provider for DB2

UDL files are normally stored in a special folder located at:

    C:\Programs Files\Common Files\System\Ole DB\data links

Microsoft Data Access Components 2.5 introduced a set of new OLE DB interfaces and functions to enumerate, create, and modify data link UDL files for configuring data sources. The `NewSnaDS.exe` utility provided as part of the OLE DB Provider for DB2 enables users to create and modify data links. This tool makes calls to the OLE DB Service Component Manager that provides these functions.

To create a new UDL file, run the `NewSnaDS` tool. This tool is installed in the System folder below the subdirectory where Microsoft Host Integration Server 2000 is installed. The default location where this tool is installed is the following:

    C:\Program Files\Host Integration Server\System\NewSnaDS.exe

A shortcut for this tool is added to the **Programs** menu under the Host Integration Server\Data Integration folder with a name of **OLE DB Data Sources**. This shortcut is created when Microsoft Host Integration Server 2000 software for the server or client (End-User Client or Administrator Client) is first installed and support for data access is selected.

A shortcut entitled the **OLE DB Data Sources Browser** is also added to the **Programs** menu in the Host Integration Server\Data Integration folder. This shortcut opens Windows Explorer to the default directory where UDL files are stored:

    C:\Programs Files\Common Files\System\Ole DB\data links

Using SNA Server 4.0 and older versions of the Microsoft Data Access Components (MDAC 2.1), it was possible to create a new

UDL file by navigating to this folder using Windows Explorer. In the right pane of Windows Explorer, right-click to open a shortcut menu and create a **New Microsoft Data Link**.

In the past, a data link file could also be created using SNA Server 4.0 with a shortcut in the SNA Server 4.0 program folder. Also, the properties of a data link file could be edited by opening the file from Windows Explorer. The procedures used with SNA Server 4.0 to create a new UDL file have been deprecated and will not work with Microsoft Host Integration Server 2000, Windows 2000, and MDAC 2.5 or later.

Once a UDL file has been created using the `NewSnaDS` tool, the file can be changed to a more appropriate name and copied to other client computers for use with the OLE DB Provider for DB2.

A new data link file can be created with the `NewSnaDS` utility using the following procedure:

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to **Data Integration**, and then click **OLE DB Data Sources** to run the `NewSnaDS` tool.
   A UDL file is created, and the **Data Link Properties** dialog box is displayed.
3. Select **Microsoft OLE DB Provider for DB2** from the list of providers, and then configure the data source information as needed.
4. Click **OK** to save the data link.

By default, data links are created in the following folder:

    C:\Program Files\Common Files\System\Ole DB\data links

However, a data link can be created in this location and moved to other client computers or folders, as needed.

Browsing Data Sources for the OLE DB Provider for DB2

By default, data links are created in the following folder:

    C:\Program Files\Common Files\System\Ole DB\data links

A shortcut is provided in the Host Integration Server program group to this folder.

1. Click the **Start** button, point to **Programs**, and then point to **Host Integration Server**.
2. Point to **Data Integration**, and then click **OLE DB Data Source Browser**.
   Windows Explorer opens in the default location where UDL files are stored. The list of data links saved in the default location appears.

Configuring Data Links for the OLE DB Provider for DB2

To edit the properties of a Data Link file, right-click the file using Windows Explorer and click **Properties**. The **Properties** dialog box appears with several property tabs:

- **General**
- **Security**
- **Summary**
- **Provider**
- **Connection**
- **Advanced**
- **All**

The **General**, **Security**, and **Summary** tabs provide access to general file information for the UDL file that is available for other files and is not related to the Data Link properties. This information includes file location, file type, file size, file dates, file security permissions for access, and descriptive summary information (description and origin properties and values such title, subject, author, and so forth) for the UDL file. The **Provider**, **Connection**, **Advanced**, and **All** tabs provide access to the Data Link properties.

The `NewSnaDS` tool can also be used to open and modify an existing UDL file. The **Data Link Properties** dialog box appears with several property tabs:

- **Provider**
- **Connection**
- **Advanced**

- **All**

Provider

The **Provider** tab enables you to select the OLE DB provider (the provider name string) to use in the UDL file from a list of possible OLE DB providers. Select the **Microsoft OLE DB Provider for DB2**. The parameters and fields displayed in the remaining tabs (**Connection**, **Advanced**, and **All**) are determined by the OLE DB Provider that is selected.

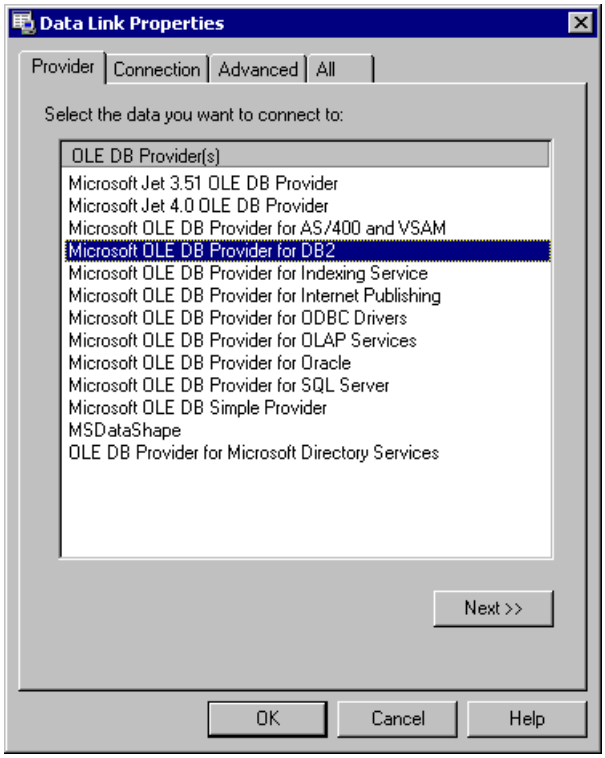


Figure 1. The Provider tab

Connection

The **Connection** tab enables you to configure the basic properties required to connect to a data source.

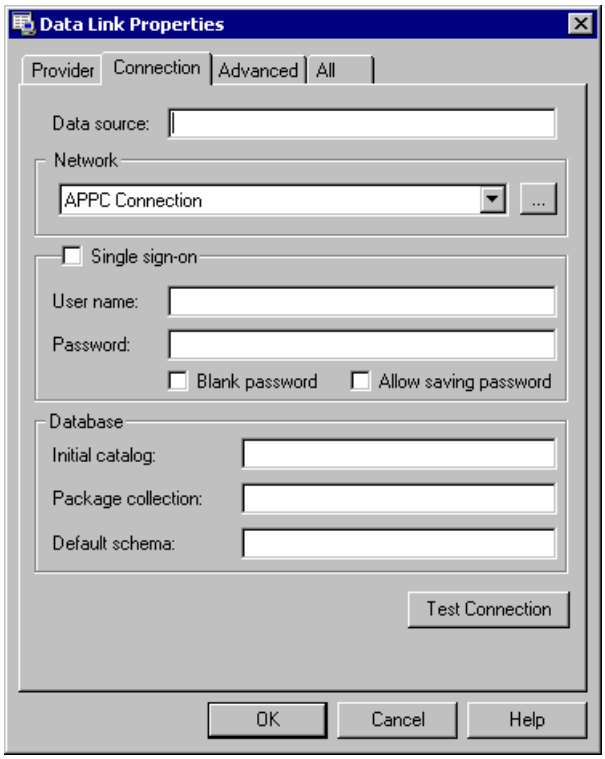


Figure 2. The Connection tab

For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following properties for Data Source and Network

connectivity values:

Property	Description
Data source	The data source is an optional parameter that can be used to describe the data source. When the `NewSnaDS` configuration program is loaded from the Host Integration Server prog ram folder, the Data source field is required. This field is used to name the UDL file, which is stored in `C:\Program Files\Common Files\System\Ole DB\data links`.
Network	This drop-down list allows you to select the type of network connection to be used. The allo wable options are **TCP/IP Connection** or **APPC Connection**. If **TCP/IP Connection** is selected, click the **More Options ...** button to open a dialog box f or configuring TCP/IP network settings. The parameters you can configure include the IP ad dress of the DB2 host (or a hostname alias for this computer) and the Network port (TCP/IP port) used for communication with the host. The default value for the Network port is 446. The IP address of the host has no default value. If **APPC Connection** is selected (using SNA LU 6.2), click the **More Options ...** button to o pen a dialog box for configuring APPC network settings. The parameters you can configure include the APPC local LU alias, the APPC remote LU alias, and the APPC mode name used f or communication with the host. The local and remote LU alias fields do not have default va lues. The default value for the APPC mode name normally defaults to QPCSUPP. The APPC mode can be selected from the drop-down list.

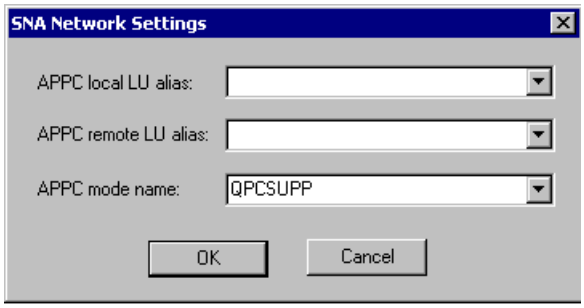


Figure 3. SNA Network Settings

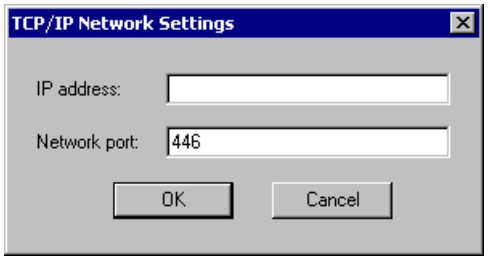


Figure 4. TCP/IP Network Settings

The **Data Source** in OLE DB is similar to a Data Source Name (DSN) in ODBC. The data source information is stored in a Microsoft Data Links file and contains the connection information required for the OLE DB Provider for DB2 to access IBM Data Base 2.

For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following properties for authentication information:

Property	Description
Single sign-on	Click this checkbox to enable using the Host Integration Security features providing a single sign-on to access this OLE DB data source. Note that single sign-on is only supported using the APPC Connection option (SNA LU 6.2). When this checkbox is selected, the User name and Password fields are grayed out and bec ome inaccessible. The user name and password fields are set based on the Windows 2000 l ogin. When this checkbox is not selected, the User name and Password fields must normally cont ain appropriate values in order to access data sources on hosts.

User name	A valid user name and password are normally required to access data sources on a host. Th ese values are case sensitive. Users must not check the **Single sign-on option** button if a specific user name and passw ord are to be entered.
Password	A valid user name and password are normally required to access data sources on hosts. The se values are case sensitive. The **Blank password** checkbox is only applicable for a Test Connection. In order to enter a password, the user will need to clear the **Blank password** check box if it is checked. If **Blan k password** is checked, then a Test Connection with a blank password will not cause the OL E DB Provider to prompt for a password. Optionally, users can choose to save the password in the UDL file by clicking the **Allow sav ing password** check box. Users and administrators should be warned that this option save s the authentication information (password) in plain text within the UDL file.

The AS/400 requires that the **User name** and **Password** parameters be in uppercase. When connecting to DB2/400, these parameters must be passed as uppercase strings. When connecting to DB2 on IBM mainframes, the **User name** and **Password** parameters can be in mixed case.

For the Microsoft OLE DB Provider for DB2, the **Connection** tab includes the following database property values:

Property	**Description**
Initial catalog	This OLE DB property is used as the first part of a 3-part fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as **LOCATION**. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to c onnect, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400, this property is referred to as **RDBNAM**. The RDBNAM value can be determine d by invoking the **WRKRDBDIRE** command from the console to the OS/400 system. If ther e is no RDBNAM value, then one can be created using the **Add** option. In DB2 Universal Database, this property is referred to as **DATABASE**. In SQL/DS (DB2/VM or DB2/VSE), this property is referred to as **DBNAME**. If the provider supports changing the catalog for an initialized data source, the consumer ca n specify a different catalog name through the **DBPROP_CURRENTCATALOG** property in t he **DBPROPSET_DATASOURCE** property set after initialization. This is a required property. This property is equivalent to the **DBPROP_INIT_CATALOG** OLE DB property ID.

Package collection	This is the name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. Package names are not restricted and can be uppercase, lowercase, or mixed case. The OLE DB Provider will create packages dynamically in the location to which the user points using the Package Collection property. By default, the OLE DB Provider will automatically create one package in the target collection, if one does not exist, at the time the user issues the first SQL statement. The package is created with GRANT EXECUTE authority to a single <AUTH_ID> only, where AUTH_ID is based on the User ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level specified when calling the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as when setting the ADO **IsolationLevel** property on the **Connection** object. A problem can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (for example, PUBLIC), then the package is created for use only by this user. This means that other users may be unable to access the required package. The solution is for an administrative user with package administrative rights to create a set of packages for use by all users (see the later section on Creating Packages for Use with DB2). The OLE DB Provider for DB2 ships with a tool program for use by administrators to create packages. The `CrtPkg.exe` tool is a Windows GUI application for use by the administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. This tool will create a set of packages and grant EXECUTE privilege to PUBLIC for all (see descriptions under the isoLevel parameter of the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as the ADO **IsolationLevel** property in the Host Integration Server 2000 online Developer's Guide). The packages created are as follows: AUTOCOMMITTED package (MSNC001 is only applicable on DB2/400) READ UNCOMMITTED package (MSUR001) READ COMMITTED package (MSCS001) REPEATABLE READ package (MSRS001) SERIALIZABLE package (MSRR001) Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400. Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE, the DB2 for OS/400 QSYS2.SYSPACKAGE, and the DB2 Universal Database (UDB) SYSIBM.SYSPACKAGE catalog tables. Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server `CrtPkg` utility to make them compatible with Host Integration Server 2000. The package names changed from SNA Server 4.0. This property is equivalent to the **DBPROP_DB2OLEDB_PACKAGECOL** OLE DB property ID.

Default schema	The name of the Collection where the OLE DB Provider for DB2 looks for catalog informatio n. The Default schema is the "SCHEMA" name for the target collection of tables and views. T he OLE DB Provider uses Default Schema to restrict results sets for popular operations, suc h as enumerating a list of tables in a target collection. For DB2, the Default schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default schema, the OLE DB Provider uses the USER _ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default value is inappropriate in many cases. T herefore it is essential that the Default schema value in the data source be defined. This property is equivalent to the **DBPROP_DB2OLEDB_CATALOGCOL** OLE DB property ID .

The **Connection** tab also includes a **Test Connection** button that can be used to test the connection parameters. The connection can only be tested after all of the required parameters are entered. When this button is pressed, an APPC session or a TCP/IP session will attempt to be established with the host using the OLE DB Provider for DB2.

Advanced

The **Advanced** tab allows users to select the character code set identifier used by the host, the PC code page used on the client, and select some specific options when using the OLE DB Provider for DB2.

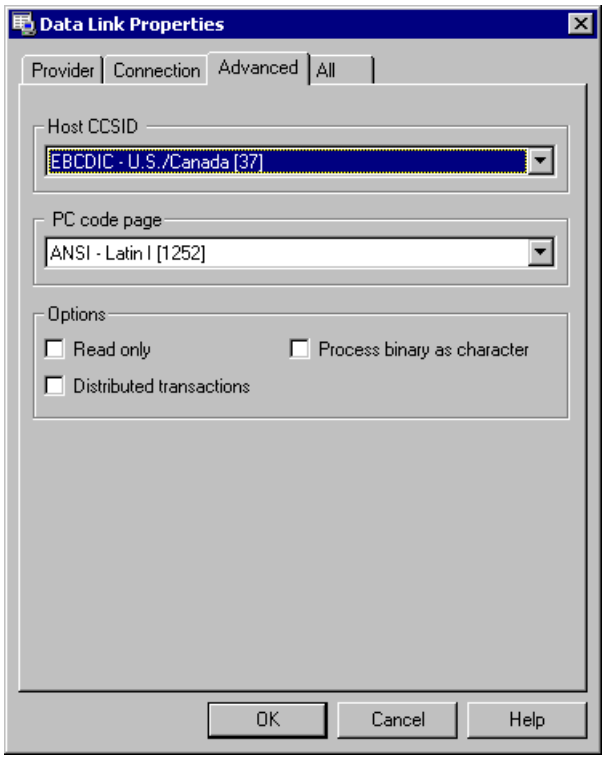


Figure 5. The Advanced tab

For the Microsoft OLE DB Provider for DB2, these properties include the following values:

Property	Description

Host CCSID	This is the character code set identifier (CCSID) matching the DB2 data as represented on th e remote host computer. The **CCSID** property is required when processing binary data as c haracter data. Unless the Process Binary as Character value is set to true, character data is c onverted based on the DB2 column CCSID and default ANSI code page. Note that Host CCSID 37 is not supported by the OLE DB Provider for DB2 when connectin g to DB2 UDB for Windows NT or DB2 UDB for AIX. This property defaults to U.S./Canada (37). This property is equivalent to the **DBPROP_DB2OLEDB_HOSTCCSID** OLE DB property ID.
PC code page	The **PC code page** property indicates the code page to be used on the PC for character cod e conversion. This property is required when processing binary data as character data. Unle ss the **Process binary as character** checkbox is selected (value is set to true), character dat a is converted based on the default ANSI code page configured in Windows. This property defaults to Latin 1 (1252). This property is equivalent to the **DBPROP_DB2OLEDB_PCCODEPAGE** OLE DB property I D.
Read only	When this option is checked, the OLE DB Provider for DB2 creates a read-only data source b y setting the Mode property to Read (DB_MODE_READ). A user has read access to objects s uch as tables, and cannot do update operations (INSERT, UPDATE, or DELETE, for example). This property defaults to a Mode property of Read/Write (DB_MODE_READ/WRITE). This property is equivalent to the DBPROP_INIT_MODE OLE DB property ID.
Process binary as character	When this option is checked (value is set to true), the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source b asis. The Host CCSID and PC code page values are required input and output parameters. This property defaults to false. This property is equivalent to the **DBPROP_DB2OLEDB_BINASCHAR** OLE DB property ID.
Distributed transactions	When this option is checked, two-phase commit (distributed unit of work) is enabled. Distri buted transactions are handled using Microsoft Transaction Server, Microsoft Distributed Tr ansaction Coordinator, and the SNA LU 6.2 Resync Service. This option works only with DB 2 for OS/390 v5R1 or later. This option also requires that an **APPC Connection** (the SNA L U 6.2 service) is selected as the Network transport in the **Connection** tab and Microsoft Tra nsaction Server (MTS) is installed. This property is equivalent to the **DBPROP_DB2OLEDB_UNITSOFWORK** OLE DB property ID.

All

The **All** tab allows users to configure essentially all of the properties for the data source except for the OLE DB Provider. The properties available in the **All** tab include properties that can be configured using the **Connection** and **Advanced** tabs as well as optional detailed properties used to connect to a data source.

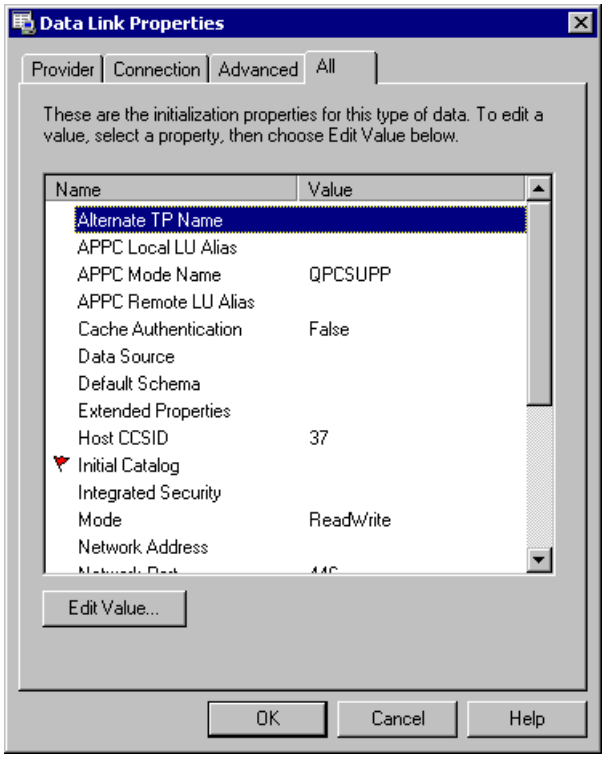


Figure 6. The All tab

The properties on the **All** tab may be edited by selecting a property from the list displayed and selecting **Edit Value**. This button will invoke a dialog box for the specific property containing a **Property Description** describing the property and a **Property Value** box for making changes.

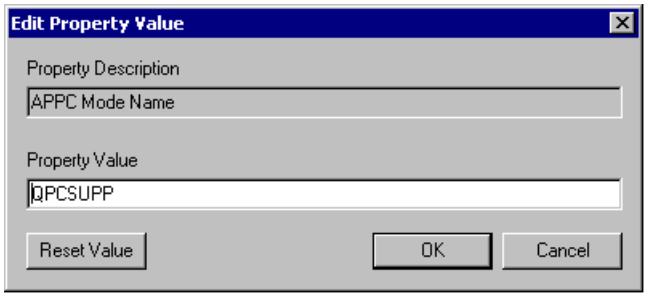


Figure 7. Edit Property Value

For the Microsoft OLE DB Provider for DB2, these properties include the following values:

Property	Description
Alternate TP Name	This is the remote transaction program name when used with SQL/DS. This property is only required when connecting to SQL/DS (DB2/VM or DB2/VSE). This property is equivalent to the **DBPROP_DB2OLEDB_TPNAME** OLE DB property ID.
APPC Local LU Alias	When an APPC Connection (SNA) using SNA LU 6.2 is selected for the Network Transport L ibrary, this field is the name of the local LU alias configured in the SNA server. This property is equivalent to the **DBPROP_DB2OLEDB_LOCALLU** OLE DB property ID.

APPC Mode Name	When an APPC Connection (SNA) using SNA LU 6.2 is selected for the Network Transport Library, this field is the APPC mode and must be set to a value that matches the host configuration and SNA server configuration. Legal values for the APPC mode name include QPCSUPP (common system default often used by 5250), #INTER (interactive), #INTERSC (interactive with minimal routing security), #BATCH (batch), #BATCHSC (batch with minimal routing security), #IBMRDB (DB2 remote database access), and custom modes. The following modes that support bi-directional LZ89 compression are also legal: #INTERC (interactive with compression), INTERCS (interactive with compression and minimal routing security), BATCHC (batch with compression), and BATCHCS (batch with compression and minimal routing security). This property normally defaults to QPCSUPP. This property is equivalent to the **DBPROP_DB2OLEDB_APPCMODE** OLE DB property ID.
APPC Remote LU Alias	When an APPC Connection (SNA) using SNA LU 6.2 is selected for the Network Transport Library, this field is the name of the remote LU alias configured in the SNA server. This property is equivalent to the **DBPROP_DB2OLEDB_REMOTELU** OLE DB property ID.
Cache Authentication	This property determines whether the OLE DB Provider caches authentication information. This property defaults to false. The value of this property (true or false) is selected from the drop-down list. This property is equivalent to the **DBPROP_CACHE_AUTHINFO** OLE DB property ID.
Data Source	The data source is an optional parameter that can be used to describe the data source. This property does not have a default value.
Default Schema	This is the name of the Collection where the OLE DB Provider for DB2 looks for catalog information. The Default Schema is the "SCHEMA" name for the target collection of tables and views. The OLE DB Provider uses Default Schema to restrict results sets for popular operations, such as enumerating a list of tables in a target collection. For DB2, the Default Schema is the target AUTHENTICATION (User ID or "owner"). For DB2/400, the Default Schema is the target COLLECTION name. For DB2 Universal Database (UDB), the Default Schema is the SCHEMA name. If the user does not provide a value for Default Schema, the OLE DB Provider uses the USER_ID provided at login. For DB2/400, the driver will use QSYS2 if there is no collection found matching the USER_ID value. Obviously, this default is inappropriate in many cases; therefore, it is essential that the Default Schema value in the data source be defined. This property is equivalent to the **DBPROP_DB2OLEDB_CATALOGCOL** OLE DB property ID.
Extended Properties	This parameter is a string containing provider-specific, extended connection information. The use of this property implies that the OLE DB consumer knows how this string will be interpreted and used by the OLE DB provider. This property should be used only for provider-specific connection information that cannot be explicitly described through the other property parameters. This property is equivalent to the **DBPROP_INIT_PROVIDERSTRING** OLE DB property ID.
Host CCSID	This is the character code set identifier (CCSID) matching the DB2 data as represented on the remote host computer. The CCSID property is required when processing binary data as character data. Unless the Process Binary as Character value is set to true, character data is converted based on the DB2 column CCSID and default ANSI code page. Note that Host CCSID 37 is not supported by the OLE DB Provider for DB2 when connecting to DB2 UDB for Windows NT or DB2 UDB for AIX. This property defaults to U.S./Canada (37). This property is equivalent to the **DBPROP_DB2OLEDB_HOSTCCSID** OLE DB property ID.

Initial Catalog	This OLE DB property is used as the first part of a 3-part, fully qualified table name. In DB2 (MVS, OS/390), this property is referred to as **LOCATION**. The SYSIBM.LOCATIONS table lists all the accessible locations. To find the location of the DB2 to which you need to c onnect, ask the administrator to look in the TSO Clist DSNTINST under the DDF definitions. These definitions are provided in the DSNTIPR panel in the DB2 installation manual. In DB2/400, this property is referred to as **RDBNAM**. The RDBNAM value can be determine d by invoking the **WRKRDBDIRE** command from the console to the OS/400 system. If ther e is no RDBNAM value, then one can be created using the **Add** option. In DB2 Universal Database, this property is referred to as **DATABASE**. In SQL/DS (DB2/VM or DB2/VSE), this property is referred to as **DBNAME**. If the provider supports changing the catalog for an initialized data source, the consumer ca n specify a different catalog name through the **DBPROP_CURRENTCATALOG** property in t he **DBPROPSET_DATASOURCE** property set after initialization. This is a required property. This property is equivalent to the **DBPROP_INIT_CATALOG** OLE DB property ID.
Integrated Security	This property determines whether the OLE DB Provider uses Host Security Integration (sing le sign-on). When this property is set to **SSPI**, single sign-on is enabled and separate user id and passw ord parameters are not required. The user id and password parameters are set based on th e Windows 2000 login. When this property is null, this single sign-on feature is disabled. This property defaults to null (host security integration is disabled) and a user id and passw ord are required. This property is equivalent to the **DBPROP_AUTH_INTEGRATED** OLE DB property ID.
Mode	A **Mode** parameter is a bit mask specifying access permissions. This bit mask can be a com bination of zero or more of the following: DB_MODE_READ—Read-only. DB_MODE_READWRITE—Read/write (DB_MODE_READ | DB_MODE_WRITE). DB_MODE_SHARE_DENY_NONE—Neither read nor write access can be denied to others. DB_MODE_SHARE_DENY_READ—Prevents others from opening in read mode. DB_MODE_SHARE_DENY_WRITE—Prevents others from opening in write mode. DB_MODE_SHARE_EXCLUSIVE—Prevents others from opening in read/write mode (DB_M ODE_SHARE_DENY_READ | DB_MODE_SHARE_DENY_WRITE). DB_MODE_WRITE—Write-only. The following values for mode are supported by the OLE DB Provider for DB2: Read (DB_M ODE_READ) and Read/Write (DB_MODE_READ/WRITE). This property defaults to **Read/Wri te**. When the **Read Only** parameter is checked in the **Advanced** tab, the OLE DB Provider for DB2 creates a read-only data source by setting the Mode parameter to **Read** (**DB_MODE_R EAD**). A user has read access to objects such as tables, and cannot do update operations (IN SERT, UPDATE, or DELETE, for example). This property is equivalent to the **DBPROP_INIT_MODE** OLE DB property ID.
Network Address	When TCP/IP has been selected for the Network Transport Library, this property indicates t he IP address of the DB2 host or a hostname alias for this computer. This property is equivalent to the **DBPROP_DB2OLEDB_NETADDRESS** OLE DB property I D.

Network Port	When TCP/IP has been selected for the Network Transport Library, this property is the TCP/IP port used for communication with the DB2 host. The default value is TCP/IP port 446. This property is equivalent to the **DBPROP_DB2OLEDB_NETPORT** OLE DB property ID.
Network Transport Library	The network transport dynamic link library property designates whether the OLE DB Provider for DB2 connects via an APPC Connection using SNA LU6.2 or TCP/IP Connection. The possible values for this property are TCP/IP or SNA. The default value for this property is SNA. If the default SNA is selected, values for APPC Local LU Alias, APPC Mode Name, and APPC Remote LU Alias are required. If TCP/IP is selected, values for Network Address and Network Port are required. This property is equivalent to the **DBPROP_DB2OLEDB_NETTYPE** OLE DB property ID.

Package Collection	This is the name of the DRDA target collection (AS/400 library) where the Microsoft OLE DB Provider for DB2 should store and bind DB2 packages. This could be same as the Default Schema. The Microsoft OLE DB Provider for DB2, which is implemented as an IBM DRDA Application Requester, uses packages to issue dynamic and static SQL statements. Package names are not restricted and can be uppercase, lowercase, or mixed case. The OLE DB Provider will create packages dynamically in the location to which the user points using the **Package Collection** property. By default, the OLE DB Provider will automatically create one package in the target collection, if one does not exist, at the time the user issues the first SQL statement. The package is created with GRANT EXECUTE authority to a single <AUTH_ID> only, where AUTH_ID is based on the User ID value configured in the data source. The package is created for use by SQL statements issued under the same isolation level specified when calling the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as when setting the ADO **IsolationLevel** property on the **Connection** object. A problem can arise in multi-user environments. For example, if a user specifies a Package Collection value that represents a DB2 collection used by multiple users, but this user does not have authority to GRANT execute rights to the packages to other users (for example, PUBLIC), then the package is created for use only by this user. This means that other users may be unable to access the required package. The solution is for an administrative user with package administrative rights to create a set of packages for use by all users (see the later section on Creating Packages for Use with DB2). The OLE DB Provider for DB2 ships with a tool program for use by administrators to create packages. The `CrtPkg.exe` tool is a Windows GUI application for use by the administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. This tool will create a set of packages and grant EXECUTE privilege to PUBLIC for all (see descriptions under the isoLevel parameter of the OLE DB **ITransactionLocal::StartTransaction** or **ITransactionJoin::JoinTransaction** methods, as well as the ADO **IsolationLevel** property in the Host Integration Server 2000 online Developer's Guide). The packages created are as follows: AUTOCOMMITTED package (MSNC001 is only applicable on DB2/400) READ UNCOMMITTED package (MSUR001) READ COMMITTED package (MSCS001) REPEATABLE READ package (MSRS001) SERIALIZABLE package (MSRR001) Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400. Once created, the packages are listed in the DB2 (mainframe) SYSIBM.SYSPACKAGE, the DB2 for OS/400 QSYS2.SYSPACKAGE, and the DB2 Universal Database (UDB) SYSIBM.SYSPACKAGE catalog tables. Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server `CrtPkg` utility to make them compatible with Host Integration Server 2000. The package names changed from SNA Server 4.0. This property is equivalent to the **DBPROP_DB2OLEDB_PACKAGECOL** OLE DB property ID.
Password	A valid user name and password are normally required to access data sources on hosts. The password is case sensitive and is displayed as asterisks in this dialog box for security purposes. This property is equivalent to the **DBPROP_AUTH_PASSWORD** OLE DB property ID.

PC Code Page	The **PC Code Page** property indicates the code page to be used on the PC for character cod e conversion. This property is required when processing binary data as character data. Unle ss the Process Binary as Character value is set to true, character data is converted based on the default ANSI code page configured in Windows. This property defaults to Latin 1 (1252). This property is equivalent to the **DBPROP_DB2OLEDB_PCCODEPAGE** OLE DB property I D.
Persist Security Info	This parameter indicates whether the data source object is allowed to persist sensitive auth entication information, such as a password along with other authentication information. Thi s property defaults to false. The value of this property (true or false) is selected from the drop-down list. This property is equivalent to the **DBPROP_AUTH_PERSIST_SENSITIVE_AUTHINFO** OLE DB property ID.
Process Binary as Character	When this property is set to true, the OLE DB Provider for DB2 treats binary data type fields (with a CCSID of 65535) as character data type fields on a per-data source basis. The Host C CSID and PC Code Page values are required input and output parameters. This property defaults to false. The value of this property (true or false) is selected from the drop-down list. This property is equivalent to the **DBPROP_DB2OLEDB_BINASCHAR** OLE DB property ID.
Units of Work	This property indicates whether two-phase commit (distributed unit of work) used for trans actions is supported for this data source. Distributed transactions are handled using Micros oft Transaction Server, Microsoft Distributed Transaction Coordinator, and the SNA LU 6.2 Resync Service. The following values for this property are supported by the OLE DB Provider for DB2: RUW (remote unit of work) DUW (distributed unit of work) Distributed unit of work (two-phase commit) works only with DB2 for OS/390 v5R1 or later . This option also requires that the SNA LU 6.2 service is selected as the network transport a nd Microsoft Transaction Server (MTS) is installed. This property defaults to RUW. The value of this property (RUW or DUW) is selected from the drop-down list. This property is equivalent to the **DBPROP_DB2OLEDB_UNITSOFWORK** OLE DB property ID.
User ID	A valid User name is normally required to access data sources on hosts. This value is case s ensitive. This property is equivalent to the **DBPROP_AUTH_USERID** OLE DB property ID.

Creating Packages for Use With DB2

The OLE DB Provider for DB2, which is implemented as an IBM Distributed Relational Database Architecture (DRDA) Application Requester, uses packages to issue SQL statements and call DB2 stored procedures. There is a provider-specific property that the OLE DB Provider for DB2 uses to identify a location in which to create and store DB2 packages. The OLE DB Provider for DB2 will create packages dynamically in the location to which the user points using the **Package Collection** property corresponding to the **DBPROP_DB2OLEDB_PACKAGECOL** property ID of OLE DB. This location may be configured using the **Connection** and **All** tabs using Microsoft Data Links or can be passed as part of the connection string as an attribute keyword and argument. This attribute keyword can be either *pkgcol* or the long form of this attribute, *Package Collection*.

There are two package creation options:

1. The OLE DB Provider for DB2 will auto-create one package for the currently-used isolation level at run time if no package

already exists. This auto-create process may fail if the user account does not have authority to create packages.

2. An administrator or user can manually creates all four packages (five packages on DB2/400) for use with all isolation levels and for use by all users (PUBLIC). The OLE DB Provider for DB2 includes a utility program for use by users with appropriate administrative privilege for this purpose.

However, some users may not have the security level when manually creating packages to GRANT authority to the packages to other users (PUBLIC, for example). This can be a problem if two or more users with different user IDs try to access a single collection of packages. The first user that created the packages will have access to the packages, but the second user likely will not. The Host Integration Server 2000 CD-ROM includes a program for use by an administrator to create packages. This tool can be run using a privileged User ID to create packages in collections accessed by multiple users. The Create Packages for DB2 utility, `CrtPkg.exe`, is a GUI-based tool included with Host Integration Server 2000 for creating packages for use with DB2. This tool is installed in the System folder below the subdirectory where the Microsoft Host Integration Server 2000 has been installed. The default location where this tool is installed is the following:

C:\Program Files\Host Integration Server\system\CrtPkg.exe

A shortcut for this tool is added to the **Programs** menu off the **Start** button on the Windows Taskbar under the Host Integration Server\Data Integration folder with a name of **Packages for DB2**. This shortcut is created when Microsoft Host Integration Server 2000 software for the server or client (End-User Client or Administrator Client) is first installed and support for Data Access is selected.

This tool will create a set of packages and grant EXECUTE privilege to PUBLIC for all:

- AUTOCOMMITTED package (MSNC001) is only applicable on DB2/400
- READ UNCOMMITTED package (MSUR001)
- READ COMMITTED package (MSCS001)
- REPEATABLE READ package (MSRS001)
- SERIALIZABLE package (MSRR001)

Note that the AUTOCOMMITTED package (MSNNC001) is only created on DB2 for OS/400.

The descriptive process name used by the `CrtPkg` utility of each package corresponds with the isolation levels defined in the ANSI SQL standard. The table below indicates how these packages correspond with the terms used by IBM for isolation levels in DB2 documentation.

Package Description	Package Name	IBM Documentation
AUTOCOMMITTED (Note that this applies only to DB2/400 and does not correspond with an ANSI SQL isolation level.)	MSNC001	COMMIT(*NONE) (NC). This isolation level is used in DB2/400 auto-commit mode only and has no corresponding isolation level on other DB2 platform s or in ANSI SQL.
READ UNCOMMITTED	MSUR001	UNCOMMITTED READ (UR). This isolation level corresponds with ANSI SQL READ UNCOMM ITTED.
READ COMMITTED	MSCS001	CURSOR STABILITY (CS). This isolation level corresponds with ANSI SQL READ COMMITT ED.
REPEATABLE READ	MSRS001	READ STABILITY (RS). This isolation level corresponds with ANSI SQL REPEATABLE RE AD.
SERIALIZABLE	MSRR001	REPEATABLE READ (RR). This isolation level corresponds with ANSI SQL SERIALIZABLE.

These Isolation Levels are described in detail under "Support for Isolation Levels Using the OLE DB Provider for DB2" in the Host Integration Server 2000 online Developer's Guide. These Isolation Levels are also described under the OLE DB isoLevel parameter and ADO **IsolationLevel** property in the Host Integration Server 2000 online Developer's Guide. Note that the AUTOCOMMITTED package (MSNC001) is only created on DB2 for OS/400.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server `CrtPkg` utility to make them compatible with Host Integration Server 2000. The package names used by the OLE DB Driver for DB2 on SNA Server 4.0 are not compatible with the OLE DB Driver for DB2 included with Host Integration Server. On SNA Server 4.0, these packages used different names as follows:

AUTOCOMMITTED package (SNANC001) only applicable on DB2/400
READ UNCOMMITTED package (SNACH001)
READ COMMITTED package (SNACS001)
REPEATABLE READ package (SNARR001)
SERIALIZABLE package (SNAAL001)

The `CrtPkg` utility will create all of these packages inside the Collection that is specified in the **Package Collection** property in the data link file, or in the connection string. If the user does not have the appropriate authority to create packages in the specified Collection, or if the specified Collection does not exist, the OLE DB Provider for DB2 will return an error.

In the case of DB2 on MVS or OS/390, the normal error text returned if the user does not have the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your
   specific DB2 version for a description of the associated Native
      Error and SQL State. SQLSTATE: 51002, SQLCODE: -567.
```

In the case of DB2/400, the normal error text returned if the user does not have the appropriate authority would be as follows:

```
A SQL error has occurred. Please consult the documentation for your
   specific DB2 version for a description of the associated Native
      Error and SQL State. SQLSTATE: 51002, SQLCODE: -805.
```

In the case of DB2/400, the normal error returned if the collection does not exist would be as follows:

```
Failed to create AUTOCOMMITTED (NC) package.  RETCODE=-99.
SQL Error: Code=-204, State=42704, Error Text= A SQL error has occurred.
   Please consult the documentation for your specific DB2 version for a
   description of the associated Native Error and SQL State.
      SQLSTATE: 42704, SQLCODE: -204
```

There are two authorities required to execute the create package process on MVS using the `CrtPkg` utility:

```
GRANT BINDADD TO <authorization ID>
GRANT CREATE IN COLLECTION <collection ID> TO <authorization ID>
```

The "authorization ID" is the user who needs the permission to create the packages. The "collection ID" is the name of the Collection, which the user specifies in the data link file for the **Package Collection** property. This Collection should be a valid Collection within the DB2.

If an administrator executes the above statements on behalf a non-privileged user, this non-privileged user can then run the `CrtPkg` utility. Once run, the `CrtPkg` process will create four sets of packages (one for each of the four isolation levels supported on DB2 for MVS or OS/390) for use by "all" (PUBLIC) users of the Microsoft data access features.

The example below illustrates this process on DB2 for MVS or DB2 for OS/390.

Grant rights to run the `CrtPkg` utility to authorization ID WNW999.

```
GRANT BINDADD TO WNW999
GRANT CREATE IN COLLECTION MSPKG TO WNW999
```

Run the `CrtPkg` utility using authorization ID WNW999 (see output from `CrtPkg` below).

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

In order to execute the `CrtPkg` utility on DB2/400, a user ID must have one of the following authorities:

- *CHANGE authority on the DB2 collection
- *ALL authority on the DB2 collection

If the user merely has *USE authority or if the user has *EXCLUDE authority, the Create Package process will fail.

There are several steps required to change user authority on a DB2/400 collection (AS/400 library): From interactive SQL (STRSQL command) while logged in as user with administrative privileges, create a new collection. This command can also be issued using ADO, OLE DB, and ODBC. However, most administrators typically create collections from the AS/400 console since the administrator must be logged in at the console to issue the Command Language (CL) command with which to change the user authority on the collection.

```
CREATE COLLECTION <collection ID>
```

From the AS/400 command console, issue the CL **WRKOBJ** command with the <collection ID> as a parameter.

```
WRKOBJ <collection ID>
```

The "collection ID" is the name of the Collection, which the user specifies in the data link file for the **Package Collection** property. This Collection should be a valid Collection within DB2. The Work with objects screen appears. Place the cursor on the *PUBLIC Object Authority line and change the authority from *USE to *ALL.

If an administrator executes the above statements on behalf a non-privileged user, this non-privileged user can then run the `CrtPkg` utility. Once run, the `CrtPkg` process will create five sets of packages (one for each of the five isolation levels supported on DB2/400) for use by "all" (PUBLIC) users of the Microsoft data access features. On DB2/400, five packages are created including the AUTOCOMMITTED packages.

The example below illustrates this process on DB2/400.

Grant rights to run the `CrtPkg` utility to authorization ID WNW999.

```
CREATE COLLECTION MSPKG
WRKOBJ MSPKG
```

Run the CrtPkg utility (see the output from CrtPkg for DB2/400 below).

```
Beginning creation process
Initializing environment...
Connecting to the host...
Connection established.
Start package creation process...
Creating AUTOCOMMITTED (NC) package...
AUTOCOMMITTED (NC) package created.
Package creation succeeded.
EXECUTE privilege on MSNC001            granted to PUBLIC
Creating READ UNCOMMITTED package...
READ UNCOMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSUR001            granted to PUBLIC
Creating READ COMMITTED package...
READ COMMITTED package created.
Package creation succeeded.
EXECUTE privilege on MSCS001            granted to PUBLIC
Creating REPEATABLE READ package...
REPEATABLE READ package created.
Package creation succeeded.
EXECUTE privilege on MSRS001            granted to PUBLIC
Creating SERIALIZABLE package...
SERIALIZABLE package created.
Package creation succeeded.
EXECUTE privilege on MSRR001            granted to PUBLIC
Free statement handles...
Disconnecting...
Disconnected
End of package creation.
Creation process has completed
```

CrtPkg allows a user to create a new UDL file or load a data source and modify an existing UDL file for connection configuration information. The **File** menu of CrtPkg has a **New** option used for creating a new OLE DB UDL File and a **Load Data Source** option to load an existing UDL file. The **File** menu **Edit Data Source** option allows a user to access and modify the properties for a data source similar to using the NewSnaDS.exe tool. The **Run** menu option is used to create packages.

When using the create package tool, if the package collection specified does not exist, then DB2 returns SQLCODE -805.

When using auto-create packages, if a package collection is not specified or the package collection does not exist, the consumer application will receive SQLSTATE HY000 and SQLCODE -385 during the "auto-create" package process. The SQLSTATE HY000 is defined as a provider-specific error. The -385 Error Return Code is not a SQLCODE but rather a DDM DRDA AR (DB2 client) return code. This error code is defined as DDM_VALNSPRM with the following associated text string:

```
"The parameter value is not supported by the target system."
```

The OLE DB Provider for DB2 client error codes are defined in the db2oledb.h file located on the Host Integration Server 2000 CD-ROM.

Note that when upgrading from SNA Server 4.0, any existing SNA 4.0 packages must be recreated using the Host Integration Server CrtPkg utility to make them compatible with Host Integration Server 2000.

SNA Server 4.0 with Service Pack 3 came with two similar utilities for creating packages: CRTPKG.EXE (a command-line tool) and CRTPKGW.EXE (a GUI-based tool).

Host Security Integration

An Integrated Security (single sign-on) feature is supported by Host Integration Server 2000 to automate the overall logon process. When configured for this feature, Host Integration Server 2000 automatically replaces special keywords in the data stream with the actual host user name and password at appropriate points in the session. This feature must be enabled by the administrator within a Host Integration Server 2000 subdomain and special strings must be entered for the user name (MS$SAME) and password (MS$SAME) that will be replaced.

When using the OLE DB Provider for DB2, this single sign-on feature works only when an APPC Connection using SNA LU 6.2 is used for the network transport. This feature is enabled under the **Connection** or **All** tabs when configuring a data source for use with the OLE DB Provider for DB2.

Troubleshooting Data Access

The Windows 2000 and Windows NT Event Viewer can be a useful tool for troubleshooting data access in some cases. The OLE DB Provider for DB2 does not issue events. However, when an APPC Connection using SNA LU 6.2 is used for the network transport for the OLE DB Provider for DB2, the low-level SNA APPC transport issues events on the SNA connection.

The Microsoft® OLE DB Provider for DB2 supplied with Host Integration Server 2000 has the ability to trace DRDA data flows when used over TCP/IP.

This DB2 tracing capability is accessible from the SNADB2 Service tracing inside the Trace utility. This facility will show the same data as an APPC trace but without the control indicators (for example, What_Received). Socket errors are traced and the error codes can be looked up in Winsock2.h supplied with the Platform SDK.

The OLE DB Provider for DB2 can return the following types of errors:

- DB2 SQL errors from the remote database
- Microsoft OLE DB Provider-specific errors
- Errors from the underlying DRDA Application Requester network client

When the OLE DB Provider for DB2 passes an error code, the best source in which to look up the meaning of the return code is often the SQL Reference or SQL Messages and Codes Reference for the target SQL database. In this case, the target database would be one of the DB2 platforms supported by the Microsoft OLE DB Provider for DB2.

The OLE DB Provider for DB2 maintains an internal integer variable named SQLCODE and an internal 5-byte character string variable named SQLSTATE used to check the execution of SQL statements on DB2. SQLCODE is set by DB2 after each SQL statement is executed. DB2 returns the following values for SQLCODE:

- If SQLCODE = 0, execution was successful.
- If SQLCODE > 0, execution was successful with a warning.
- If SQLCODE < 0, execution was not successful.
- If SQLCODE = 100, "no data" was found. For example, a FETCH statement returned no data because the cursor was positioned after the last row of the result table.

SQLSTATE is also set by DB2 after the execution of each SQL statement. Application programs can check the execution of SQL statements by testing SQLSTATE instead of SQLCODE. SQLSTATE provides application programs with common codes for common error conditions (the values of SQLSTATE are product-specific only if the error or warning is product-specific). Furthermore, SQLSTATE is designed so that application programs can test for specific errors or classes of errors.

SQLSTATE values consist of a two-character class code value, followed by a three-character subclass code value. The first character of an SQLSTATE value indicates whether the SQL statement was executed successfully or unsuccessfully (equal to or not equal to zero, respectively). Class code values represent classes of successful and unsuccessful execution conditions. The following SQLSTATE class codes are used by DB2:

Class Code	Description of Error Class
00	Successful completion. Execution of the SQL statement was successful and did not result in any type of warning or exception condition.
01	Warning
02	No data
07	Dynamic SQL error
08	Connection exception
0A	Feature not supported
0F	Invalid token
21	Cardinality violation
22	Data exception
23	Constraint violation
24	Invalid cursor state
25	Invalid Transaction State
26	Invalid SQL statement identifier
2D	Invalid transaction termination

34	Invalid cursor name
39	External function call exception
40	Transaction rollback
42	Syntax error or access rule violation
44	WITH CHECK OPTION violation
51	Invalid application state
53	Invalid operand or inconsistent specification
54	SQL or product limit exceeded
55	Object not in prerequisite state
56	Miscellaneous SQL or product error
57	Resource not available or operator intervention
58	System error

The SQLSTATE value of HY000 is defined as a provider-specific error. An SQLSTATE of 08S01 (connection exception with a subclass code of S01) also indicates a provider-specific error. This means the SQLCODE should be looked up in the driver-specific documentation included with the OLE DB Provider for DB2.

If the SQLSTATE does not indicate a driver-specific error when the OLE DB Provider for DB2 passes back an SQLSTATE of 08S01, it indicates a network error. For example, an SQLCODE of -603 is a provider-specific error that is mapped to DB2OLEDB_COMM_HOST_CONNECT_FAILED in the db2oledb.h include file supplied with the OLE DB Provider for DB2. Errors with an SQLSTATE of 08S01 are documented in the db2oledb.h include file (the SQLCODE value) which is located on the Host Integration Server 2000 CD-ROM in the SDK\Include subdirectory.

The following steps are useful in researching an error. Start by reading the provided error text returned by the OLE DB Provider for DB2. In some cases, the error text provides very limited useful information. For example, error text from an SQLCODE of -603 states the following:

> Test connection failed because of an error in initializing provider.
> Could not connect to specified host.

The next step is to lookup the SQLSTATE to determine the source of the error. Is the error a DB2 error, a network client error, or an OLE DB Provider error? An SQLSTATE of 08S01 is defined as follows:

> Communication link failure.

This definition is intended to inform the user, administrator, or developer that the error is related to the OLE DB Provider's underlying network client.

Unfortunately, many of the SQLSTATE codes returned by the OLE DB Provider for DB2 are DB2 errors and are not documented in the OLE DB Provider for DB2 on-line help.

The SQLSTATE of HY000 is defined as a provider-specific error. An SQLSTATE of 08S01 also indicates a provider-specific error. This means the SQLCODE should be looked up in the provider-specific documentation included with the OLE DB Provider for DB2.

If the SQLSTATE does not indicate a driver-specific error, then the SQLCODE should be looked up in the appropriate DB2 manual for the target platform. For example, an SQLCODE of -603 is documented in Appendix B, SQLCODEs and SQLSTATEs, in the *AS/400 Advanced Series DB2 for AS/400 SQL Programming, Version 4*, Document Number SC41-5611-00 published by IBM. An SQLCODE of -603 corresponds to SQLSTATE 23515 in the DB2 for OS/400 error code list. For example, the explanation for this SQLCODE is as follows:

> Unique index cannot be created because of duplicate keys.

When the SQLSTATE and the SQLCODE definitions documented in these appendices create a mismatch with the actual errors returned, this usually indicates a provider-specific error condition.

A final step to understand an error is to check the db2oledb.h file. This file is not installed by the Host Integration Server or Host Integration Client setup program, but can be found on the Host Integration Server 2000 product CD ROM in the SDK\Include subdirectory. An SQLCODE (for example, -603) can be looked up by searching the right-most column of the db2oledb.h file for a value near to 603. In this case, one will see a comment "/* -600 */" and can then count down three additional lines to line number 603. The internal error code -603 is defined as follows:

> DB2OLEDB_COMM_HOST_CONNECT_FAILED

Unfortunately, this error text is not further defined anywhere in the software or documentation provided to the customer. This

particular error usually indicates a problem with the configuration parameters or the connection string passed.

Integrating RPG and CL Programs Using the Microsoft OLE DB Provider for DB2

Microsoft Corporation

July 2003

Applies to:
Host Integration Server 2000, Microsoft® OLE DB Provider for DB2

Summary: This document describes how to use DB2 stored procedures to call RPG and CL programs (running on an IBM iSeries or AS/400 system) with the Microsoft OLE DB Provider for DB2. Samples of RPG and CL server programs are provided, as well as sample Microsoft Visual Basic® 6, Microsoft Visual Basic .NET, and Microsoft Visual C#® .NET client-side code to connect to those programs.

Download the article.

Microsoft Host Integration Server 2000 Product Overview

Microsoft Corporation

October 2001

Summary: This white paper offers an overview of the integration components provided by Microsoft Host Integration Server 2000. These components enable data integration, application integration, and network integration of host-based systems with Microsoft .NET platform-based applications. (31 printed pages)

Contents

Introduction

An estimated 70 percent of all corporate data is stored on host systems, such as IBM mainframe and AS/400 computers. Yet, increasingly, organizations rely on personal computers together with Web-based and Windows®-based applications for everyday productivity and line-of-business solutions. Companies have discovered that Web and Windows solutions often are easier to learn and quicker to implement than comparable host-based applications. To preserve their time and capital investments in host technology, organizations must either migrate all of their host-based resources to the Windows platforms, which can be expensive and time-consuming, or integrate their host-based resources with more efficient Windows-based and Web-based solutions.

Integrating host-based data and applications with Web-based and Windows-based applications offers significant benefits, including:

- Preserves investment in currently deployed host and PC technology while taking advantage of new architectures and products being offered for the PC platform.
- Allows rapid deployment of custom, high-performance solutions, using a choice of Windows-based development tools and leveraging a large pool of qualified developers who do not need to know or learn host programming.
- Lowers administrative resources and reduces hardware expenses, thereby reducing the total cost of ownership (TCO).

Whether companies want to create data warehouses to improve decision-making, develop Web-based applications that perform transactions using host-based data, or allow users to include archived data in reports, Microsoft Host Integration Server 2000 offers integration components that make it easy to achieve those goals.

To help its customers achieve these benefits, Microsoft has offered a host integration solution since 1990, when it introduced Communication Server 1.0 in partnership with Digital Communications Associates. Microsoft SNA Server 2.0, which followed in 1992, allowed system administrators to send local area network (LAN) and SNA networking traffic across the same network infrastructure.

Since then, Microsoft has continued to improve SNA Server based on customers' needs, developing it into a complex and feature-rich product. Host Integration Server 2000 builds on the strengths of SNA Server 4.0 and offers a range of mature technologies that help companies solve their host integration challenges.

Today's Web Solutions

Organizations frequently need to integrate their host systems with Web-based applications. This is why Microsoft offers products and technologies today that developers can use to build and deploy applications for the business Internet, which includes high traffic e-commerce Web sites, corporate intranets and enterprise supply chain integration. These products and key building technologies include the following:

- Internet Information Services 5.0 (IIS) in Windows 2000 with Active Server Pages (ASP), Microsoft Commerce Server 2000, Microsoft BizTalk™ Server 2000, Microsoft Internet Security and Acceleration Server 2000, and Microsoft Application Center 2000, for deploying dynamic, scalable, and secure Web sites.
- Microsoft Message Queuing (MSMQ) in Windows 2000 for reliable asynchronous transactions.

- Microsoft SQL Server™ 2000 for developing high performance Web stores and data warehouses.
- COM+ component and programming model for developing applications using popular development tools, such as the Microsoft Visual Studio® development system.
- Using Host Integration Server 2000 in combination with these products and technologies, companies can create highly manageable, scalable and reliable distributed applications that use existing host-based resources. Host Integration Server 2000 components and services all work together; they all share a common programming model, component model and tools and are designed to work with each other. This allows developers to focus on business problems, not systems integration. By tightly combining many of the common "plumbing" services into the Windows 2000 operating system and providing access points for the Microsoft Visual Studio development system to those services, developers can spend more time on building reusable business logic components and less on underlying maintenance code that is common to and necessary for all applications.

Microsoft .NET

Microsoft recognizes that today's Web largely resembles mainframe systems, where vital data is locked up in centralized servers designed to publish information in often-predetermined HTML pages. Decision makers and knowledge workers are granted limited, often read-only, access to vital data, with little or no opportunity to interact with or edit this data using Web browsers.

Microsoft is creating an advanced new generation of software that melds computing and communications in a revolutionary new way, offering developers the tools they need to transform the Web and every other aspect of the computing experience. Microsoft .NET will allow the creation of distributed Web Services that integrate and collaborate with a range of complementary services to make information available any time, any place and on any device.

The fundamental idea behind Microsoft .NET is that the focus is shifting from individual Web sites or devices connected to the Internet, to constellations of computers, devices and services that work together to deliver broader, richer solutions. People will have control over how, when and what information is delivered to them. Computers, devices and services will be able to collaborate with each other to provide rich services, instead of being isolated islands where the user provides the only integration. Businesses will be able to offer their products and services in a way that lets customers seamlessly embed them in their own electronic fabric. It is a vision that extends the personal empowerment first offered by the PC in the 1980s.

Microsoft .NET will help drive a transformation in the Internet that will see HTML-based presentation augmented by programmable XML-based information. XML is a widely supported industry standard defined by the World Wide Web Consortium, the same organization that created the standards for the Web browser. It was developed with extensive input from Microsoft Corp. but is not a proprietary Microsoft technology. XML provides a means of separating actual data from the presentational view of that data. It is a key to the Next Generation Internet, offering a way to unlock information so that it can be organized, programmed and edited; a way to distribute data in more useful ways to a variety of digital devices; and a way of allowing Web sites to collaborate and provide a constellation of Web Services that will be able to interact with each another.

Microsoft .NET comprises the following:

- **Microsoft .NET platform**
  Includes .NET infrastructure and tools to build and operate a new generation of services; .NET User Experience to enable rich clients; .NET building block services, a new generation of highly distributed mega services; and .NET device software to enable a new breed of smart Internet devices.
- **Microsoft .NET products and services**
  Includes Windows Server 2003, with an integrated set of building block services; MSN .NET; personal subscription services; Office .NET; and Visual Studio .NET.
- **Third-party .NET services**
  A vast range of industry partners and developers will have the opportunity to produce corporate and vertical services built on the .NET platform.

Microsoft .NET will take computing and communications far beyond the one-way Web to a rich, collaborative, interactive environment. Powered by advanced new software, yet built on today's .NET Enterprise Server products, Microsoft .NET will harness a constellation of applications, services and devices to create a personalized digital experience—one that constantly and automatically adapts itself to your needs and those of your family, home and business. It means a whole new generation of software that will work as an integrated service to help you manage your life and work in the Internet Age.

For consumers, that means the simplicity of integrated services; unified browsing, editing and authoring; access to all your files, work and media online and off; a holistic experience across devices; personalization everywhere; and zero management. It means, for example, that any change to your information—whether input via your PC or handheld or smart credit card—will instantly and automatically be available everywhere that information is needed.

For knowledge workers and businesses, it means unified browsing, editing and authoring; rich coordinated communication; a seamless mobile experience; and powerful information-management and e-commerce tools that will transparently move between

internal and Internet-based services, and support a new era of dynamic trading relationships.

For independent software developers, it means the opportunity to create advanced new services for the Internet Age—services that are able to automatically access information either locally or remotely, working with any device or language, without having to rewrite code for each environment. Everything on the Internet becomes a potential building block for this new generation of services, while every application can be exposed as a service on the Internet.

The Microsoft .NET vision means empowerment for consumers, businesses, software developers and the entire industry. It means unleashing the full potential of the Internet. And, it means the Web the way you want it.

The foundation for the Microsoft .NET platform is based on the Microsoft .NET Enterprise Server products, including Windows 2000, Microsoft SQL Server 2000, and Host Integration Server 2000.

Host Integration Server 2000 Components

Integration projects are as varied as the companies that undertake them, and Host Integration Server 2000 includes a wide array of integration components and tools to help create an effective integration solution.

Host Integration Server 2000 provides the following categories of components:

- **Data Integration components**, which provide desktop or server-based applications with direct access to host data. Host Integration Server 2000 provides a comprehensive set of data access services, which includes direct data access to relational and non-relational mainframe and AS/400 data through open database connectivity (ODBC), object linking and embedding database (OLE DB), and COM automation controls.
- **Application Integration components**, which allow host-based and Web- or Windows-based applications to communicate directly with one another. Host Integration Server 2000 delivers solutions for integrating both synchronous and asynchronous transactions.
- **Host Integration Server 2000 Management components**, which provide a wide assortment of tools to manage the components of Host Integration Server 2000. This includes tools for performing both interactive and scripted local and remote Web-based and traditional client/server management of Host Integration Server components.
- **SNA Network components**, which connect SNA networks with PC-based LANs. Host Integration Server 2000 allows users running Windows, Macintosh, UNIX, the MS-DOS® operating system, and IBM OS/2 to share resources on mainframes and AS/400 systems without requiring system administrators to install resource-heavy SNA protocols on the PCs or install costly software on the host.

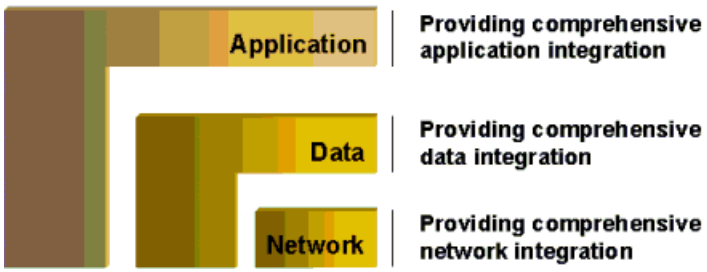


Figure 1. Host Interoperability Layers. Host Integration Server 2000 offers a comprehensive set of components for the Network, Data, and Application integration layers with which to integrate Windows and the .NET platform with host systems.

Host Interoperability Layers

When thinking of the many services that Host Integration Server 2000 provides, it is useful to think of these services in groups or "layers"—similar to the network protocol stack: Network Data, Application, and Management. In the following sections of this paper, we will explore these layers in more detail, beginning with Data, then Applications, and finally Network (and Management).

The Data Integration Layer

The *Data Integration layer* of Host Integration Server 2000 provides access to both structured and non-structured data stored on IBM mainframe or AS/400 computers. This data can be stored in a database or file system. In addition to data access, the Data Integration layer is also responsible for providing data transfer services between Windows 2000 computers and host systems. The Data Integration layer consists of components that make use of existing mainframe and AS/400 software.

The Data Integration layer can be broken down further into the following categories:

- Relational database access
- Record file access

- File transfer
- AS/400 data queue access

All these services make use of IBM host-based products that implement *the IBM Distributed Data Management Architecture* (DDM). DDM is a framework or methodology for sharing and accessing data between systems. DDM defines the "how to communicate" and leaves it up to individual platform vendors to implement the DDM architecture. IBM currently supports DDM for most IBM platforms, including: OS/390 (MVS), AS/400, RS/6000 (AIX), and AS/36. By supporting DDM, application developers are freed from having to write complex communications interfaces for each platform they need to support. Instead the application and DDM handles this complexity on behalf of the application.

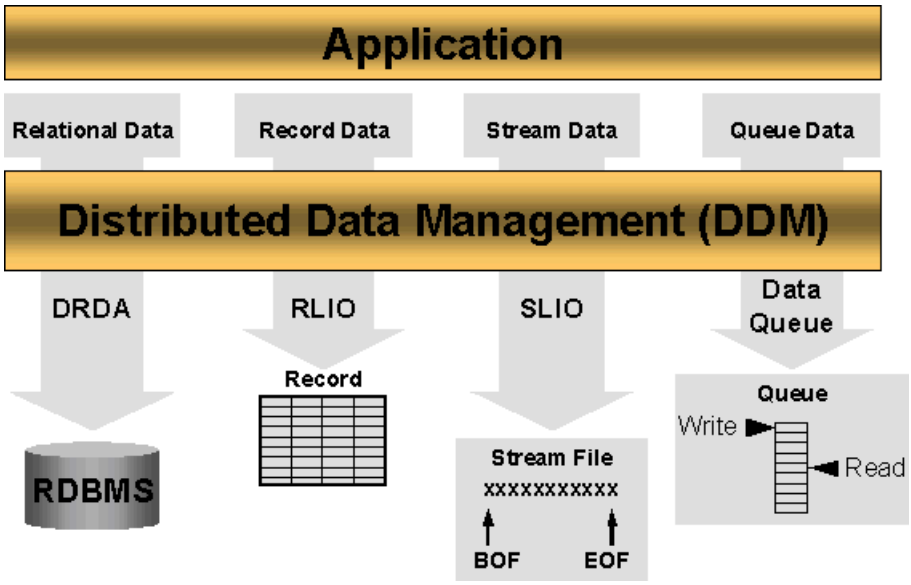


Figure 2. Distributed Data Management. Components of the Host Integration Server 2000 Data Integration layer utilize popular DDM file models when integrating host data sources with Windows and .NET applications.

Host Integration Server 2000 offers relational database access by using the *Distributed Relational Data Architecture* (DRDA) subset of DDM, non-relational access using the *Record Level I/O* (RLIO) implementation of DDM, while file transfer and AS/400 data queue access employ a subset of the RLIO protocol.

Relational Database Access

Much of the operational data stored on OS/390, AS/400, and RS/6000 computers is accessed via a relational database management system. The most popular database on these host systems is IBM DB2. In the case of the AS/400, DB2 is integrated with the operating system. For OS/390 and RS/6000 computers, it is common for organizations to deploy the IBM DB2 relational database management system (RDBMS).

What all of these host systems have in common is that data stored in these databases are accessible as relational tables using Structured Query Language (SQL). This allows for efficient and standardized access to the data on the local DB2 system. However, for many years, there was no common means of accessing data across systems on remote DB2 computers. To resolve this problem, IBM devised Distributed Relational Databases Architecture (DRDA) and has passed the architecture to The Open Group for publication and future extension.

DRDA offers both Remote Unit of Work (RUW) and Distributed Unit of Work (DUW) access to host data. RUW is used for read-only and simple updating of database tables using SQL statements and stored procedures. DUW is used when updates span multiple DB2 instances or computer systems and supports the two-phase commit (2PC) protocol. The 2PC protocol ensures that changes to multiple databases will either succeed or fail in their entirety. We will talk more about 2PC and transaction management in the section on the Application Layer later in this white paper.

Through its Universal Data Access (UDA) architecture, Microsoft supports two popular methods of accessing remote relational databases: the industry-standard Open Database Connectivity (ODBC); and the broader Object Linking and Embedding DB (OLE DB). ODBC is designed specifically for interoperating with SQL-accessible RDBMSs. ODBC is implemented by independent software vendors (ISVs) in the form of either a back-end data base *driver*, or as a front-end application (e.g., reporting or query tool). Microsoft and other vendors offer ODBC drivers for most of the popular RDBMSs. Microsoft defined OLE DB as a multi-tier distributed architecture for accessing both SQL RDBMSs and non-SQL data sources (e.g., mail folders, Internet server stores, flat file systems). In the OLE DB architecture, ISVs develop software that participates in one of three roles: (1) OLE DB *provider*, or back-end data source driver, (2) OLE DB *service component* (e.g., query processor, cursor engine), and (3) OLE DB consumer (e.g., Web service or application, GUI query or reporting tool). OLE DB is based on the Component Object Model (COM) and OLE DB providers are designed to expose a well-known set of interfaces. When a provider cannot expose specific, useful or often-required functionality, an OLE DB service component is employed to extend and standardize the abilities of the provider. In this way, OLE

DB consumers can be written to access multiple data sources without knowing any of the vagaries or limitations of a given back end provider.

Host Integration Server 2000 implements access to DB2 via two features:

- Microsoft ODBC Driver for DB2
- Microsoft OLE DB Provider for DB2

The first of these methods is the ODBC Driver for DB2. It relies on an underlying DRDA application requester (AR) developed by Microsoft. The DRDA AR connects the ODBC driver to DB2 on popular platforms, including OS/390, OS/400, RS/6000-AIX, and Windows NT®, Windows 2000.

It provides a flexible way for developers using the ODBC API to create applications that can access DB2 records quickly and efficiently. The driver supports the DRDA Level 3 standard and ODBC 3.x interfaces, and allows application programmers to write C and C++ applications that issue dynamic SQL queries and call DB2 stored procedures.

The second method to access DB2 is through the OLE DB Provider for DB2. This component is also implemented to sit on top of the DRDA AR, and therefore supports the same target DB2 systems and substantially the same DB2 access features (e.g., dynamic SQL and stored procedures, 2PC, SNA LU6.2 and TCP/IP network connectivity). Developers can use C or C++ to integrate DB2 data with Web-based and Windows-based applications. Visual Basic® and Web developers (using scripting languages such VBScript) can use the higher-level ActiveX® Data Objects (ADO) to develop e-commerce solutions. Additionally, DB2 is directly accessible from productivity applications, such as Microsoft Office 2000 using Visual Basic for Applications (VBA) and ADO from within Excel.

Many organizations want to improve corporate decision making by centralizing data that is stored in a variety of formats in a number of different places. Database administrators can use *Data Transformation Services* (DTS), a feature of Microsoft SQL Server 2000 and Microsoft SQL Server 7, to import and export data between multiple heterogeneous sources using the OLE DB Provider for DB2. Using this tool, administrators can create a data warehouse using DB2 data, plus integrate most other data sources accessible via an OLE DB provider.

The *Distributed Query Processor* (DQP), another feature of Microsoft SQL Server, allows users to access data that resides on multiple, distributed databases across multiple servers. Using DQP, SQL Server administrators and developers can create linked server queries that run against multiple back-end data sources with little or no modification. DQP enables application developers to create heterogeneous queries that join tables in SQL Server with tables in DB2. Also, DQP can be used to create SQL Server views over DB2 tables so that developers can write directly to SQL Server and integrate both Windows-based and host-based data in their applications with ease.

Record File Access

Another rich source of legacy information is the large amount of data still stored in mainframe VSAM files, Partitioned Datasets, and AS/400 files. Host Integration Server 2000 supports the following services for access to non-relational host data:

- The OLE DB provider for AS/400
- The OLE DB provider for VSAM

The OLE DB Provider for AS/400 supports record level access to keyed and non-keyed physical files with external record descriptions, as well as logical files with external record descriptions. Also, the provider can use an optional Host Column Description (HCD) file to describe the format of the target file, mapping the AS/400 data types to OLE DB data types, allowing the developer to access AS/400 flat data files and source files.

The OLE DB Provider for VSAM, which relies on the HCD files to define the metadata of the target data set or member, provides access to most types of mainframe based VSAM files.

Sequential Access Method (SAM) data sets

- Basic Sequential Access Method (BSAM) data sets
- Queued Sequential Access Method (QSAM) data sets

Virtual Storage Access Method (VSAM) data sets

- Entry-Sequenced Data Sets (ESDSs)
- Key-Sequenced Data Sets (KSDSs)
- Fixed-length Relative Record Data Sets (RRDSs)
- Variable-length Relative Record Data Sets (VRRDSs)
- VSAM Alternate Indexes to ESDSs or KSDSs

Basic Partitioned Access Method data sets

- Partitioned Data Set Extended (PDSE) members
- Partitioned Data Set (PDS) members
- Read-only support for PDSE directories
- Read-only support for PDS directories

Using Visual Studio, developers can build dynamic Web applications that integrate host non-relational data sources with Windows data, allowing knowledge workers to publish needed information for use by their organization's decision makers.

File Transfer

Most 3270 emulators support the ability to transfer files between a mainframe computer and a workstation using the IND$FILE utility program. This program works in conjunction with a host operating system such as TSO or teleprocessing monitor software such as CICS running on the mainframe. This process, is often manual and is somewhat inefficient due to the need to use 3270 terminal emulation on the client and to have the host operating system act as an intermediary in the data transfer process. Host Integration Server 2000 provides several more efficient methods to perform file transfer. These methods are:

- Host File Transfer
- APPC File Transfer Protocol (AFTP)
- AS/400 Shared Folders

The *Host File Transfer* utility lets developers move files between a host system and a local Windows computer. Host Integration Server 2000 provides this service through a single ActiveX Control. This extends the ability of the client application to perform file transfer operations from a large number of client development environments. Using HCD files, the Host File Transfer can access the same mainframe data set types as the OLE DB Provider for VSAM, yet it is optimized to download or upload the entire contents of the data set or member. Other supported environments include the AS/400 and AS/36.

The TCP/IP based File Transfer Protocol (FTP) is often used to move files between computer systems running under UNIX, VMS, and other operating systems. This capability is typically provided as a utility program that implements a set of commands that can be used to connect to a remote computer, log on, navigate to specific locations in the local and remote computer file systems, and then transfer a file (or multiple files) to or from that computer. Unfortunately, to use this protocol to transfer files to a host computer would require TCP/IP on the host. (Most data center managers are reluctant to support TCP/IP on a host computer due to security and performance issues.) Because of the popularity of this protocol, however, IBM has implemented a similar SNA function, the *APPC File Transfer Protocol* (AFTP). This allows files to be transferred between SNA systems using commands that are so similar to FTP commands that anyone familiar with FTP can easily use AFTP to perform file transfer functions. Internally, AFTP transfers files using the LU 6.2 program-to-program protocol, which is quite efficient for transferring files. AFTP software can be installed either on the Host Integration Server 2000 server or client and used to transfer files to an SNA host.

The AS/400 *Shared Folders* feature of Host Integration Server 2000 allows a Windows NT or Windows 2000 administrator to re-share a file on an AS/400 host as if it is a local file system directory. Because the AS/400 shared folders feature uses standard operating system file sharing, it requires no software on the client. The client simply sees the folder as a standard Windows NT or Windows 2000 shared directory. This feature is implemented in Host Integration Server 2000 using the same AS/400 PC Support software that allows workstations to access AS/400 files in a pure SNA network configuration.

AS/400 Data Queue Access

AS/400 *Data Queues* are used on an AS/400 to send data records between separately executing programs. Multiple AS/400 client programs can send data records to a single server program running on an AS/400. Alternatively, a single client program can send records to an AS/400 Data Queue and multiple server programs can extract the records and process the data in parallel. This feature proved so useful in developing AS/400 applications that IBM extended the use of AS/400 Data Queues to PC workstations. Host Integration Server 2000 enables Windows 32-bit applications to access data queues via the AS/400 Data Queue COM Automation Control. Host Integration Server 2000 lets developers access AS/400 data queues from a PC running Windows, so they can move part or all of their AS/400 applications from an AS/400 computer to a PC platform and still use the PC-based program to access a remote data queue on the AS/400.

Application Integration Layer

The Application layer provides the services that enable Windows and mainframe programs to work together. It defines how two application programs can participate in cross platform transactions and messaging, including distributed database updates involving the two-phase commit database protocol.

In Host Integration Server 2000, the Application layer is focused on distributed application programs where at least one of the programs is running on a mainframe and AS/400.

Integrating existing host-based data and applications with Web-based solutions can benefit organizations in many ways. This allows you to preserve your investments in existing technology while extending host-based resources to highly scalable, distributed, component-based and Web-based applications. It helps you to reduce your development costs by allowing you to draw on a large pool of qualified developers for Windows rather than a small group of host programmers with highly specialized skills. It also supports cutting migration costs by keeping host-based resources on mainframe and AS/400 computers or amortizing these costs by migrating slowly to the PC platform over time.

Transaction Processing

In mainframe applications, database management systems and transaction processing programs have always been tightly integrated. In applications developed using IBM's CICS or IMS, a communications front-end program normally retrieves data from and/or updates a back end database. Multiple interactions with the user are often necessary to complete a transaction and update a database. In the case where multiple database updates are required, then CICS and IMS support the concept of a *Transaction*. In the case of multiple updates to a database, the scope of a transaction ensures that all updates are successfully applied or that any partially completed updates are reversed. A CICS or IMS program can also indicate the success or failure of a transaction by issuing a command to indicate the end of a transaction or that a rollback of partially completed updates is required.

Mainframe transaction processing systems also support the definition of a transaction that can exist across multiple distributed databases. This feature is based upon a well-known standard called *Synchronization Level 2* (Sync Level 2) also known as the Two-Phase Commit protocol.

CICS in particular can be used to write distributed applications where one CICS application program can link to another CICS program whether that program is on the same computer or a different one. CICS uses a special CICS data area, called a *commarea*, to pass data to the target program. It is the CICS commarea that plays the key part in passing data to and from the linked-to program.

When developing transaction-aware applications for the Windows and Web platforms, developers often write COM components that run under the *Microsoft Transaction Server* (MTS) in Windows NT or the equivalent COM+ feature in Windows 2000. MTS combines the features of a COM object broker and a transaction manager. An application can be written with COM components that run on different computers. The programmer can define how each component participates in transactions across these distributed components. Each COM component can specify whether it can be part of a new transaction or an existing transaction. The transaction management part of MTS is based on a component called the *Distributed Transaction Coordinator* (DTC). Once the transaction state of a set of components is defined, then MTS can use the DTC to enforce the same transaction and database update integrity over them that a CICS or IMS program can enforce over mainframe transactions. COM programs can indicate the success or failure of a complete transaction just as a CICS or IMS program can do.

In order to allow applications for Windows NT and Windows 2000 to make use of these pre-existing mainframe CICS and IMS programs, Host Integration Server 2000 offers *COM Transaction Integrator* (COMTI). COMTI enables mainframe CICS and IMS programs to participate in COM transactions.

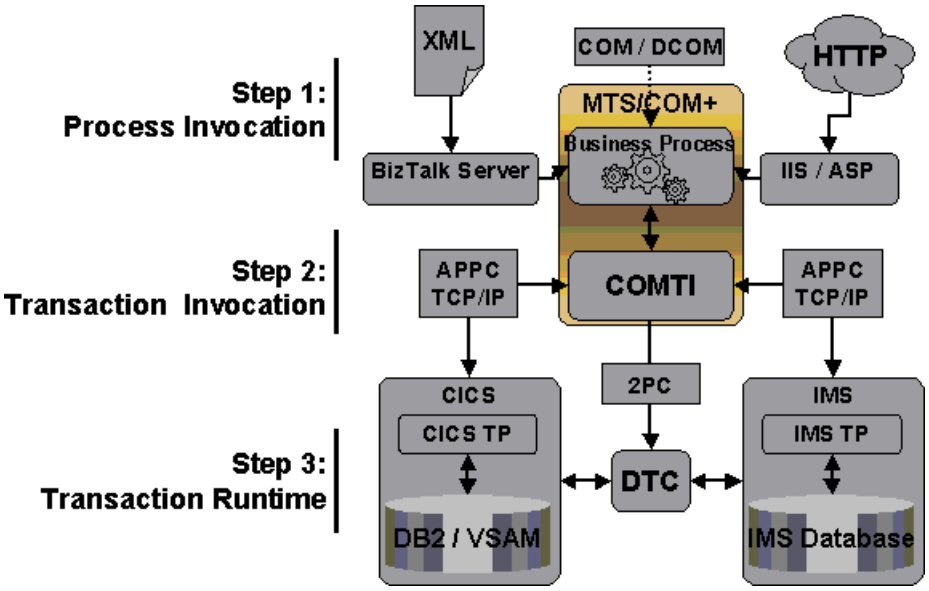


Figure 3. COM Transaction Integrator. COMTI allows developers to preserve existing CICS and IMS environments while moving to a new Windows and .NET platform.

COMTI creates a COM+ or MTS "wrapper" (or proxy) component that makes a legacy CICS or IMS mainframe application program look like a COM component at execution time. Windows 2000 applications make method calls and pass parameters to what appears to be just another COM component. This call is translated under the covers to the appropriate CICS program call and sent to the mainframe via Host Integration Server 2000 and LU 6.2. As part of this process, the COMTI component converts

parameters between mainframe and COM formats. COMTI transactions can also participate in cross platform transactions under the control of the Distributed Transaction Coordinator.

Using COMTI is a three-step process:

1. Building the COMTI component
2. Adding the component to MTS
3. Running the application

First, the COBOL commarea description must be brought down to the developer's workstation from the mainframe using Host File Transfer or a terminal emulator. Next, the COMTI graphical user interface is used to import the COBOL data description and convert the COBOL parameter definitions into their COM equivalents. Next, COMTI will generate the COM component that will act as a proxy for the mainframe application program. The commarea is converted into a COM recordset definition that can be used to pass parameters into and out of the mainframe program. After the code is translated, the name of the program becomes the callable method of the MTS transaction and the variables passed to the original COBOL program in the commarea are translated into method parameters.

After the COMTI component is created, it can be packaged with other COM components to define the complete transaction. At execution time, calls to legacy programs appear to be calls to the COMTI component. Parameters are transparently converted and any required two-phase commit protocols are enforced across both systems.

Currently, COMTI uses the *Distributed COM* (DCOM) protocol to communicate between the COM components and Host Integration Server 2000. (Host Integration Server 2000 takes care of the conversion to the appropriate SNA protocol.)

Inter-Platform Message Queuing

In the section on Transaction Processing, we also discussed how Host Integration Server 2000 supports keeping remote databases in strict synchronization with each other using distributed transactions and COMTI. When synchronous transactions are required, COM and COMTI provide an excellent real-time method of implementing a distributed application.

There is a third scenario in which programs need to pass data between them, but where there is no need for the sending program to wait for the receiving program to process the data. In this case it is enough to know that the data will be delivered and processed eventually. Message passing software to support this capability is called Message Oriented Middleware (MOM). Microsoft's MOM product offering is the **Microsoft Message Queuing System** (MSMQ).

This system consists of agents running on the sending systems and message queues on the receiving system. A program that wants to send a message to a receiving system simply uses an API to place the message in a local send queue. Eventually the message is sent. Although the initiating program does not wait for the message to be successfully received by the recipient, it can assume that the message will be delivered. This is because the MOM system uses transactional integrity between the local sending agent and the receiving program.

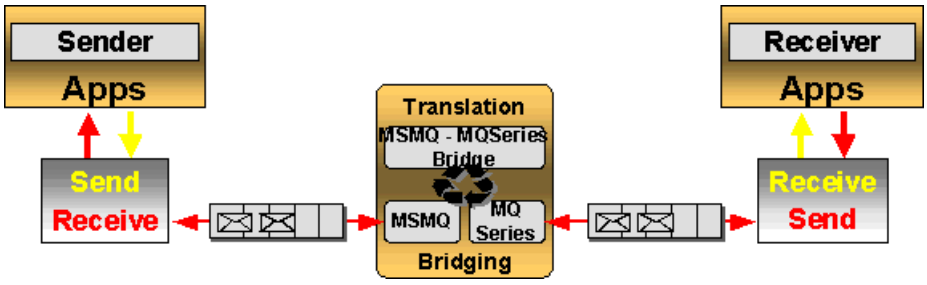


Figure 4. MSMQ-to-MQSeries Bridge. The bridge provides asynchronous, messaging-based, communication integration between heterogeneous applications.

Microsoft's MSMQ is the MOM product that supports messaging between Windows platforms. Mainframes and other platforms, on the other hand, normally use a product developed by IBM called MQSeries. IBM has extended MQSeries to other IBM and non-IBM platforms in addition to mainframes and AS/400. Although a version of MQSeries is available for Windows NT and Windows 2000, MSMQ is native to those platforms. In order to support cross platform messaging between Windows and mainframe messaging systems, Host Integration Server 2000 includes the MSMQ-MQSeries Bridge. This bridge integrates the two messaging platforms and enables messages to be transferred in either direction across platforms.

Management Layer

In the preceding sections of this paper we discussed how Host Integration Server 2000 client and server components work together to provide access to mainframe and AS/400 resources. In this section we will cover the Management layer of Host Integration Server 2000. This layer concerns itself with how Host Integration Server 2000 servers and clients are installed and managed. It also discusses the integration of Host Integration Server 2000 with mainframe network management systems such

as IBM NetView.

Host Integration Server 2000 Server Installation

The Host Integration Server 2000 server installation allows the administrator to specify the components to be installed and the various roles that a server will play such as primary or backup configuration server. Server installation can be performed via several methods such as:

- Local CD-ROM
- Network share point
- Unattended installation
- Systems Management Server

The default method of installing Host Integration Server 2000 is from a local CD. This type of installation is run at the server, and the installer must respond to all prompts.

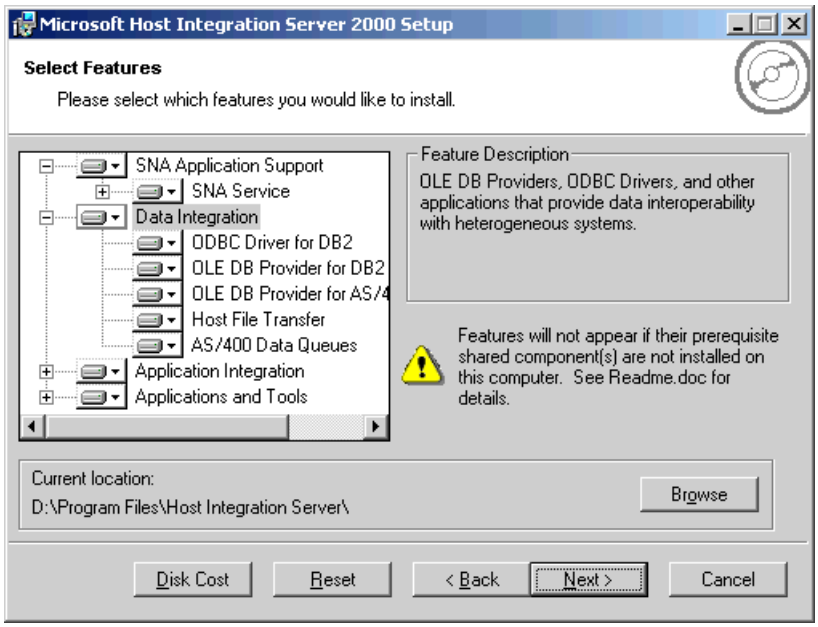


Figure 5. Server Setup. Host Integration Server 2000 offers a Microsoft Software Installer setup program that allows the administrator to choose the most appropriate features.

Alternatively, the installation files can be located on a network share point. The installer must still answer all the questions during the installation process.

Host Integration Server 2000 also supports an unattended install process. This process uses answer files that remove the need for the installer to be present during the complete installation. They still need to be there to start the process, however. This type of install must be done from a network share point or diskettes since it requires that the installation files be customized. Once the process is started the installer can leave.

With Systems Management Server an installer does not even have to go to the server to start installation of Host Integration Server 2000. Systems Management Server can be used to push the server installation and configuration to a target server.

Host Integration Server 2000 Server Configuration

Once the server is installed it must be configured. The configuration process is used to define all the link services, connection types, LUs and LU pools. It is also used to define users groups and client workstations and allocate resources to them. Resources can be allocated globally to everyone or specifically down to a particular user, group or even a single workstation. Resource access can also be granted at different levels of granularity such as to a subdomain, a specific server, LU pools, or even specific LUs.

Host Integration Server 2000 Client Installation

The Host Integration Server 2000 CD also includes installable clients for Windows 98, Windows 3.x, Windows NT Workstation and Windows 2000 Professional. It also includes installation instructions or the source of clients for several other platforms.

There are three parts to each Host Integration Server 2000 client—not all of which need to be installed. These parts are the Base client, API support, and features such as the OLE DB providers, 3270 and 5250 clients. Most third party emulators only require that you install the Base support.

When installing Host Integration Server 2000 client, the installer can specify the components to install, select the client/server protocols that the client uses to communicate with the server, and define how the client will locate a server in the network. The methods used to install a client are similar to the ones used to install a server with one important new addition, the Web based install.

These methods are:

- Local CD-ROM
- Network share point
- Unattended installation
- Systems Management Server
- Web based installation

The first three methods are essentially the same. Just as in the case of Host Integration Server 2000 servers, an answer file can be used to simplify the installation process. In the case of the Systems Management Server installation there are also Systems Management Server Installation Packages on the CD for Windows 3.1, Windows 98, Windows NT Workstation, and Windows 2000 Professional.

With Web based installation, a Host Integration Server 2000 client can be installed by simply displaying an intranet or Internet Web page and clicking on a hypertext link. The Web-based installation provides three options. The installer can request the download and installation of the 3270 Client plus the full Host Integration Server 2000 Client, the 5250 Client plus the full Client, or just the Base Client. (The latter is useful in order to upgrade an older Base Client that came with a third party emulator.) This process can also be customized to provide additional capabilities.

Host Integration Server 2000 Client Configuration

Once the client is installed it must be configured. This configuration is normally limited to specifying how the client will locate a sponsor server when it connects. Options are to have the client broadcast a request for a sponsor to the entire subdomain, or to direct its request to a specific server or list of servers.

Host Integration Server Network Resource Management

The predecessors of Host Integration Server 2000 have always been the industry leaders in Windows to host network integration using standard networking protocols. Host Integration Server 2000 carries forward this tradition by enhancing support for a wide variety of SNA links, connections and logical unit types. Host Integration Server 2000 includes tools to manage the setup and use of these resources in a secure, loosely, or tightly controlled fashion. For instance, individual servers, Logical Units or Logical Unit pools can be set up to grant access to any client that requests access, or they can be restricted to specific users and/or client workstations.

The security mechanism for this access control is based on Windows NT and Windows 2000 security. User and Groups can be defined and Access Control Lists created for Host Integration Server 2000 resources that will allow or deny these users, groups or computer systems access.

Host Integration Server 2000 Management Tools

There are several administrative interfaces available to be used to manage Host Integration Server 2000 resources. These tools include:

- Microsoft Management Console
- Scripting
- Web Based Administration
- SNA Remote Access Services
- Mainframe tools

The *Microsoft Management Console* (MMC) is the standard Microsoft tool for assembling individual systems management utilities into customized toolsets that can be used to delegate administrative tasks and responsibilities to individual administrators. Host Integration Server 2000 installation provides *snap-ins* that can be added to any MMC console. Note that multiple Host Integration Server 2000 snap-ins can be installed in a console in order to manage multiple subdomains or servers.

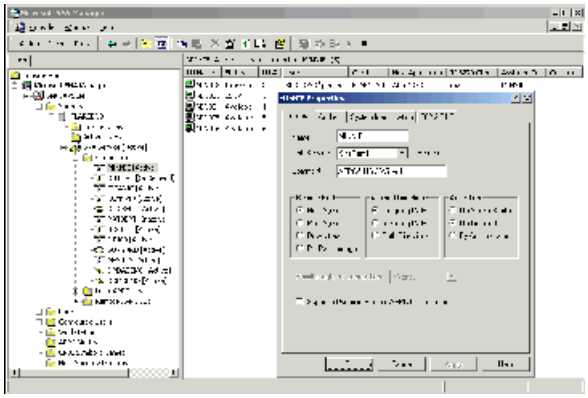


Figure 6. Server configuration (click to enlarge). Host Integration Server 2000 offers an MMC-based snap-in for administering server configurations.

As an alternative to the GUI based MMC, Host Integration Server 2000 also comes with command line utilities that can be used to manage any Host Integration Server 2000 resource. The Windows Script Host is a powerful scripting language that is available as an add-on to Windows NT and comes standard with Windows 2000. The addition of this tool to the Host Integration Server 2000 command line utilities enables the administrator to create and modify any Host Integration Server 2000 resource from a batch script. Using a combination of the Windows Script Host and the SNACFG command line utility, any Host Integration Server 2000 management task can be automated and performed without resorting to the GUI.

Host Integration Server 2000 also supports administration using Web-based tools using Microsoft's Windows Management Instrumentation (WMI) technology. This is an implementation of the Desktop Management Task Force's (DMTF) Web-Based Enterprise Management (WBEM) initiative for Microsoft Windows platforms.

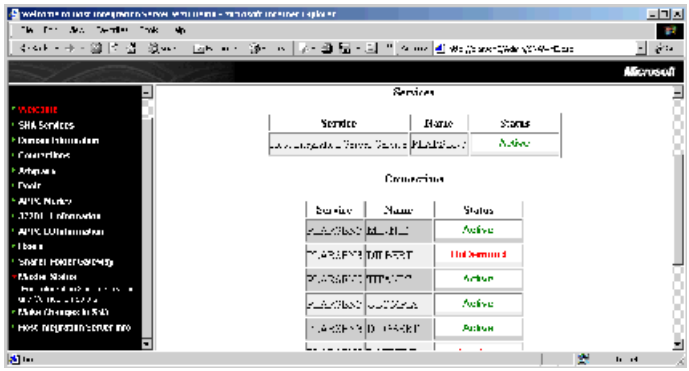


Figure 7. Web-based Administration. Microsoft Host Integration Server 2000 provides enhanced management capabilities for SNA administrators to view and control their SNA Server environment. Using WMI/COM, administrators can build their own Web-based tools to monitor, configure, and administer all SNA objects.

Normally an administrator will be able to access a server or subdomain using the client/server protocols that we have been discussing in this paper. In some cases, however, this may not be possible. *SNA Remote Access Services* (SNARAS) extends normal Windows NT and Windows 2000 RAS to operate over SNA links. This allows an administrator to connect to a remote SNA server over an SNA link.

In addition to integrating with Windows NT and Windows 2000 management tools, Host Integration Server 2000 also integrates with popular mainframe-based network management tools such as IBM *NetView*. Windows NT and Windows 2000 Event Log Alerts can be forwarded up to a mainframe NetView system using the NVAlert component. A NetView operator can also run Host Integration Server 2000 commands using the NVRunCmd command provided by Host Integration Server 2000. In addition, the NetView Response Time Monitor (RTM) can be used to capture information on client response time and forward the data back to NetView.

Active Directory Integration

Host Integration Server 2000 also takes advantage of the Windows 2000 Active Directory™ service. A client needing to locate a sponsor server when it connects to the network can use Active Directory. It can also use Active Directory to locate a server that can provide a specific Host Integration Server 2000 resource such as an LU or LU pool.

Host Integration Server 2000 uses Active Directory by registering services and resources with the Active Directory schema. The benefits of using Active Directory include:

- Client configuration and resource location on the network is simplified.
- A permanent LAN connection between Host Integration Server 2000 and Host Integration Server 2000 clients is not needed.

- The limitation of 8,000 sponsor connections that existed in SNA Server 4.0 is eliminated.

Host Integration Server 2000 client computers must be configured to communicate to a Host Integration Server 2000 computer using either sponsor connections or Active Directory. A client computer cannot be set up to use both at the same time. Host Integration Server 2000 extends the Active Directory schema to include Host Integration Server resources.

Host Security Integration

One of the problems with mixing Windows NT and Windows 2000 with mainframe and AS/400 computers is that each type of system has its own way of dealing with security. It is not uncommon to have one user account and password to access a local Windows NT or Windows 2000 domain while also having another user account and password to access the mainframe or AS/400. In addition, mainframe and/or AS/400 applications may also have their own user account and password. After a while, users begin to forget theses multiple passwords and begin to write them down and keep them in an insecure place. This defeats the purpose of having passwords in the first place.

To simplify the situation, Host Integration Server 2000 comes with a Host *Security Integration* feature, which consist of a set of processes that run on network servers to provide the following services:

- User account and password mapping and caching
- Password Synchronization
- Single Sign on to multiple domains and host security systems

The *Host Account Cache* is a database that maps Host and Windows NT/Windows 2000 users names and passwords. It supports two options: Replicated and Mapped.

The *Replicated* option is used where the user's name and/or password are the same on both platforms. The *Mapped* option is used where they are different. This information is used to translate between the local and remote user name and passwords.

The *Host account synchronization service* is used to intercept Windows NT and Windows 2000 password changes and send them to a host security system.

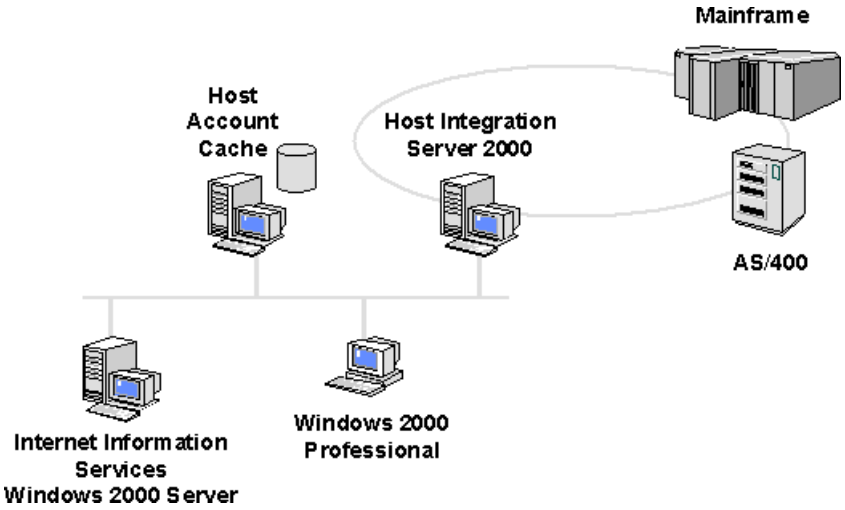


Figure 8. Host Security Integration. Host Integration Server 2000 allows administrators to synchronize the account information on the host with the Windows 2000 domain. Synchronized accounts allow for single sign-on, meaning that users only have to maintain a single user account and password to log on to Windows and the host system.

In the case of AS/400, password synchronization works in both directions. In the case of mainframes, a third party tool is necessary for complete bi-directional synchronization with mainframe security systems such as RACF, ACF/2 and Top Secret.

Network Integration Layer

Traditionally Windows NT and Windows 2000 networks used Microsoft Networking that was based on NetBIOS. Most recently, TCP/IP has become the preferred network protocol used in Windows NT and Windows 2000 networks. In fact, with Windows 2000 the favored protocol is TCP/IP.

The most popular method of connecting networks with dissimilar protocols has always been to insert a gateway device between them that can convert from one protocol to another. To bridge the gap between a Windows 2000 network and an SNA network requires an SNA gateway. Host Integration Server 2000 is the most popular method of providing this function. Host Integration Server 2000 incorporates the traditional SNA gateway function, however, it goes beyond simple protocol conversion by providing a significant number of higher level application enabling services that work in conjunction with it's basic function.

The *Network layer* contains all the low level protocols used to transport data between the client and the Host Integration Server 2000 and between the Host Integration Server 2000 and the host computers in the SNA network. It also includes support for 3270 and 5250 terminal emulation as discussed below.

SNA-to-PC LAN connectivity sits at the heart of every integration project, and Microsoft is committed to providing companies with efficient, flexible SNA gateway options. Host Integration Server 2000 builds on the industry leading network connectivity features of SNA Server 4.0 to provide companies with the greatest number of options for connecting SNA systems and PC-based networks.

Host Integration Server 2000 connects PC-based LANs to IBM System/390 mainframe and AS/400 midrange systems. With Host Integration Server 2000, users with desktop operating systems such as Windows 2000 Professional, Windows NT Workstation, Windows 9x, Macintosh, UNIX, MS-DOS and IBM OS/2 can share resources on mainframe computers and AS/400 systems. Administrators can provide this connectivity in many cases without installing SNA protocols on the PC or deploying software on the host system. The following diagram illustrates the network connectivity capabilities of Host Integration Server 2000.

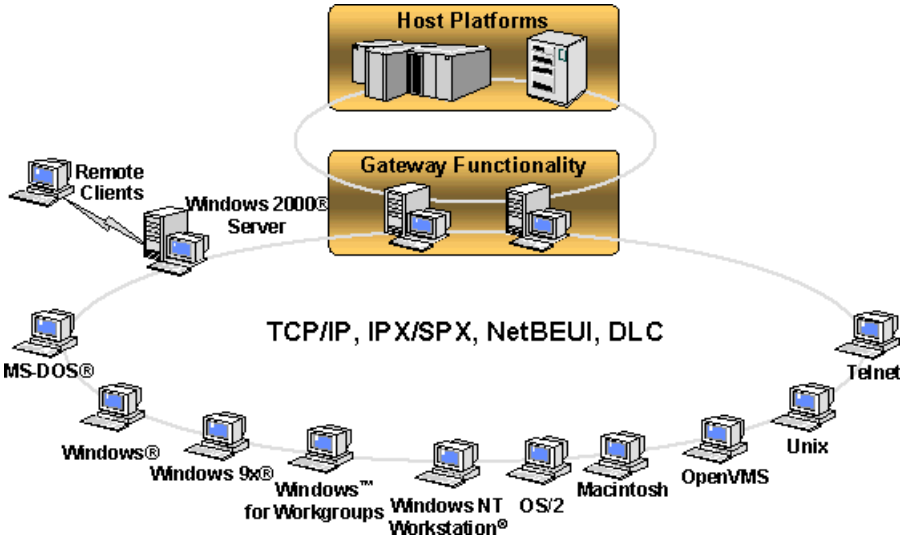


Figure 9. Host Integration Server 2000 Network Support. Host Integration Server 2000 builds on Microsoft SNA Server's strong platform of protocol and connectivity options. As an SNA gateway solution, Host Integration Server 2000 runs on Windows NT Server and Windows 2000 Server to connect PC-based local area networks (LANs) to IBM System/390 mainframe and AS/400 midrange systems.

Host Integration Server 2000 Software Architecture

Host Integration Server 2000 has been designed with a modular Client/Server architecture.

Server architecture

The Server component of Host Integration Server 2000 runs on Windows NT 4.0 or Windows 2000 Server and provides gateway and protocol conversion services. It contains most of the SNA protocol stack and is also responsible for notifying the attached clients of the configuration of the servers, available services, and network status. It consists of the following components:

- SNASERVER
- SNABASE
- SNADMOD

SNASERVER runs as a Windows NT or Windows 2000 Service. It is the protocol conversion engine of Host Integration Server 2000. SNASERVER provides PU 2 and 2.1 node emulation. It communicates status changes with other server, clients and host computers. It also allows the SNA Manager to display the status of Links, Connections, and LUs in the network. It uses the following components to communicate between clients and other servers.

SNABASE runs as a service on the server. It is responsible for building and maintaining a list of available servers, links, LUs and invokable Telecommunication Programs (TPs). It also sends this list to other Host Integration Server 2000 servers and receives updates from those servers. It maintains this information in a local configuration file (COM.CFG).

When a client first contacts the Host Integration Server 2000 server, SNABASE uses this information to determine the server actually hosting the desired resources for which the client is looking. Then the server will send the client the list of available servers holding the desired resource; this initial connection is called a Sponsor connection (see below).

After connection, SNABASE directs the client to the LU or LU pool that can provide the client with the desired resource.

SNADMOD is a common communications module used by the server to communicate with clients and other servers. It uses a remote Procedure Call (RPC) protocol that is proprietary to Host Integration Server 2000.

Client architecture

The architecture of the Host Integration Server 2000 client includes:

- APIs
- SNABASE (or equivalent)
- SNADMOD

The Host Integration Server 2000 client provides APIs needed by SNA applications such as 3270 emulators and APPC applications.

The architecture of Host Integration Server 2000 clients differs depending on whether the Host Integration Server 2000 client is for Windows NT, Windows 2000 or another operating system. The Windows NT and Windows 2000 clients also use SNABASE below the API layer. Other clients use a callable DLL instead of a service to perform the same functions, but all the clients use DMOD to implement common communications between clients and servers.

Initially Host Integration Server 2000 clients are configured to attempt to contact any server in the subdomain (see below), or a list of specific Host Integration Server 2000 servers. If the configuration specifies a list of servers, then one will be chosen at random each time a client attempts to connect. This design causes requests to be spread across multiple servers, in effect implementing an efficient form of load balancing across the servers. This design also provides a degree of fault tolerance. If a server or resource is unavailable, then the client can automatically choose a resource from another server. This load balancing and fault tolerance is inherent in the design of Host Integration Server 2000 and does not have to be added using any external form of Windows NT or Windows 2000 clustering or fault tolerance.

Host Integration Server 2000 servers are organized in logical groups called *Subdomains* (to distinguish them from Windows NT/2000 Domains). There can be any number of Host Integration Server 2000 Subdomains. Each Subdomain can contain up to 15 Host Integration Server 2000 servers whose roles in a Host Integration Server 2000 network can be defined as *Primary* Host Integration Server 2000 servers, *Backup* Host Integration Server 2000 servers or *Member* Host Integration Server 2000 servers. These designations are somewhat analogous to the Primary, Backup and Member server designation of Windows NT and Windows 2000 Servers. In this case, however, instead of ownership of the security accounts database, Host Integration Server 2000 server roles define ownership of a configuration database that describes the state of all servers, their controlled resources, and the status of connected clients.

All three types of servers can host resources for clients to use. A new client looking for resources can contact all three types of servers. Primary servers can also update the configuration database. Backup servers, on the other hand, have only a read-only copy of the database for local use. Member servers cannot provide a client with information on network resources since they do not have a copy of the configuration database. They can, however, host specific resources, such as LUs, for clients to use.

Status information is replicated between Host Integration Server 2000 servers to allow all Host Integration Server 2000 servers to maintain a configuration file that describes the complete status of all servers and other Host Integration Server 2000 managed resources in the network.

When a client computer connects to the network, it contacts a server near it based on it's own configuration file. This connection is called a Sponsor Connection. The sponsor server can provide the client with a list of resources in the network or refer it to another server in the network. From this connection the client retrieves the status of existing servers and other network resources such as individual Logical Units or Logical Unit pools that the client can request.

The Client component of Host Integration Server 2000 is a fairly slim software component since most of the protocol conversion and enabling services take place on the server. A wide range of clients are supported, including Windows 2000, Windows NT, Windows 98/95, Windows 3.X, MS-DOS, and OS/2. In addition, Host Integration Server 2000 client software is available for Macintosh, UNIX and VMS clients via third parties.

Server Deployment Models

Another important aspect of the Host Integration Server 2000 network architecture is how Host Integration Server 2000 servers and clients can be deployed in a network. Using Host Integration Server 2000, organizations can deploy SNA gateways in an enterprise network in one of three deployment models:

- Centralized Deployment model
- Branch Deployment model
- Distributed Deployment model

Centralized Deployment model

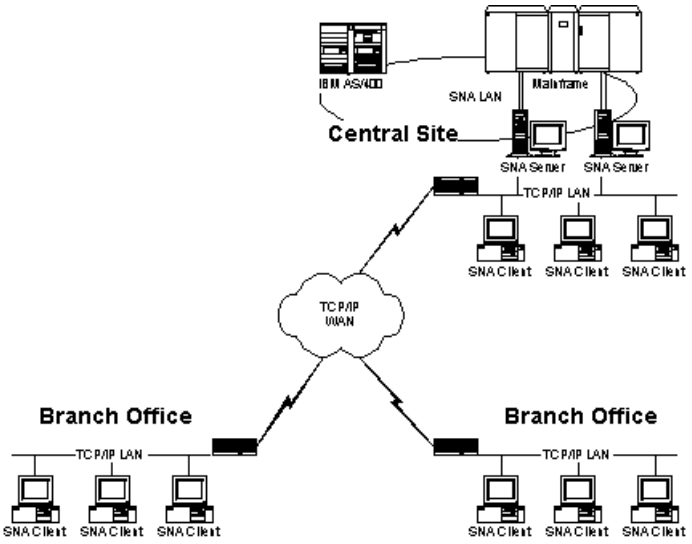


Figure 10. Centralized Deployment model. In a centralized deployment, Host Integration Server 2000 computers support local and remote split-stack SNA clients, server-based applications, and TN3270 emulators.

The *Centralized Deployment model* locates the Host Integration Server 2000 servers at the data center near the host computer in the network. In this model the SNA gateways are located at the data center and connect to the host using native SNA protocols, such as a high-speed token-ring or channel attachment. In the centralized deployment model, Administrators can isolate the SNA traffic to the data center to avoid supporting SNA traffic on the WAN. Centralized SNA gateways provide split-stack or TN3270 service for local and remote desktops that connect to the gateways using standard LAN protocols.

The major advantage of this model is that the Host Integration Server 2000 server has high-speed access to the host computer via a common LAN and front-end processor, or even direct channel attachment. This location offers many advantages; because the servers are located in a single location, clients can take advantage of the designed-in load balancing and hot backup capabilities provided by multiple Host Integration Server 2000 servers. In addition, MIS personnel can service and administer the centralized servers.

In the case of an organization with clients located at remote branch offices, several disadvantages that must be considered. Because most of the protocol conversion is performed on the server, there will be quite a lot of low-level data traveling back and forth between the clients and servers. One of the other two models may be more appropriate in these cases.

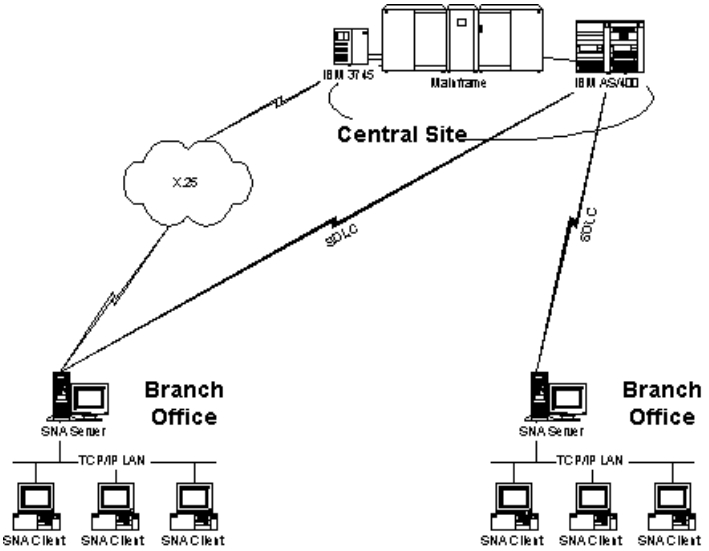


Figure 11. Branch-based Deployment model. Host Integration Server 2000 supports the traditional way to deploy SNA gateways in branch offices, via SDLC leased lines and X.25 connections to the host.

Branch Deployment model

The *Branch Deployment model* places the SNA gateways close to the clients at branch offices. The advantage of this location is that the heaviest data traffic will occur between the clients and the servers over the local LAN connection at the branch office. Only highly compressed and relatively efficient SNA traffic will occur over the Wide Area Network between the branch office and the central site.

Another advantage of this organization is that local branch personnel can administer the servers located at the branch. (This can be viewed as an advantage or a disadvantage depending on whether you want the servers managed centrally or remotely.) One of the enhancements of Host Integration Server 2000 over its predecessor, SNA Server 4.0, is that Host Integration Server 2000 servers can easily be administered remotely. This makes non-centralized Host Integration Server 2000 servers less of an issue compared with SNA Server 4.0.

A disadvantage of this organization is that there is no high-speed connection between the server and the host. WAN connections and routers in this type of organization will have to be sized to handle the traffic between the servers and the hosts. Because servers are not centrally located, it is also not possible to take advantage of the load balancing and hot backup features of Host Integration Server 2000.

Distributed Deployment model

The *Distributed Deployment model* combines the best aspects of the previous two deployment models at a slight increase in setup and administration complexity. With this model Administrators can deploy Host Integration Server 2000 concurrently in the branch and central sites. The branch-based Host Integration Server 2000 computers provide client-to-server support and connect to the centralized computers running Host Integration Server 2000 using native TCP/IP, or IPX/SPX protocols. The centralized servers provide server-to-host support and connect to the host with a high-speed token ring or direct channel attachment using native SNA protocols.

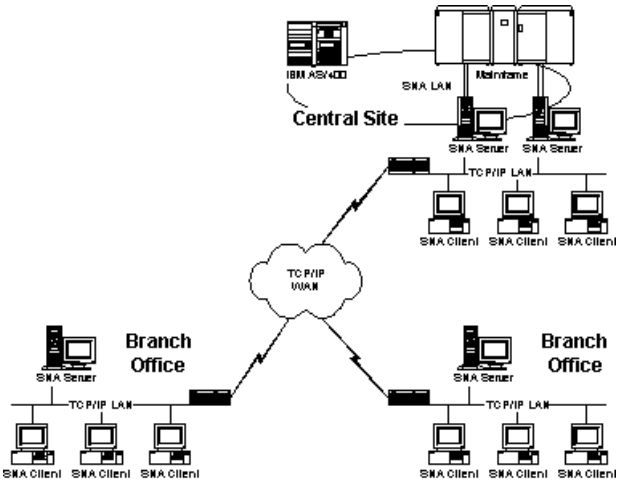


Figure 12. Distributed Deployment model. Host Integration Server 2000 offers a distributed deployment model that makes better use of available WAN bandwidth than centralized or branch-based configurations.

In this configuration the links between the central server (or server pool) and the host computer is a fast LAN to Front End Processor or direct channel attachment. The links between the clients and their local server (or servers) can also be a fast local LAN connection. Only relatively compact SNA traffic travels between the remote and central servers. This retains the advantage of pooling Host Integration Server 2000 servers at the central site while giving remote clients a local sponsor connection and local LU pools.

The only disadvantage of this approach is that there are more servers to deploy, administer and maintain. This is not a serious drawback, however, because they can be administered from a central site if necessary using the Host Integration Server 2000 remote administration capability. This model is a little more expensive because more servers will have to be deployed. This is not a serious impediment, however, because Host Integration Server 2000 servers run on modest hardware.

In general the Distributed Deployment model is more flexible and cost-effective than the Branch-based or Centralized models. Compared to a Branch-based deployment, the Distributed approach eliminates the time-outs associated with bridging or tunneling 802.2 using routers, improves host response times and simplifies network management. Compared to a Centralized deployment, the Distributed approach decreases WAN usage and supports NetView management of the Branch-based servers.

Of course the preceding discussion of deployment models assumes that the host is a single mainframe or AS/400 computer located at a central site. It is also possible that the network includes more than one host computer and/or that host computers and clients are distributed at multiple sites. Here, too, the Distributed Deployment Model can be used to combine the best of the Centralized and Branch organizations by placing some of the Host Integration Server 2000 servers near their hosts for high speed access, and placing some of the servers near the clients that they service. For example, assume that two companies merge, and that each company has a community of users and their own host computer. Eventually each company will require access to its own host computer and that of its partner. In this case the Distributed Deployment Model can be used at both companies to give symmetric access to both hosts from clients at both companies.

In addition to these deployment models, Host Integration Server 2000 can also act as a proxy for other network attached SNA devices that need to access a host computer through Host Integration Server 2000. This is supported by Host Integration

Server 2000 *Downstream Connection support and PU passthrough connections*. These capabilities may be useful in some specialized cases.

Network Protocols

Host integration Server 2000 supports a wide variety of protocols for communicating between clients, the server, and host computers. This allows Host Integration Server to fit into whatever networking architecture you currently have in place.

Server to host protocols

Host Integration Server 2000 supports several protocols for communicating between the server and the host computer. Link Services—provided by Microsoft, IBM, and other third parties and installed in the server—implement these Link Services.

For mainframe host access the most popular protocols include:

- Channel attachment
- LAN connection via the DLC 802.2 protocol
- Synchronous Data Link Control (SDLC)
- X.25

Direct channel attachment is used to connect centralized Host Integration Server 2000 servers to a host using Escon or Bus and Tag channel connection methods. LAN Attachment via DLC 802.2 is used to connect servers to hosts over Ethernet, Token Ring, or Fiber Distributed Data (FDDI) LANs. Synchronous Data Link Control (SDLC) is used to connect them over wide area dial-up or leased line connections. X.25 is used to connect Host Integration Server 2000 server and mainframe over an X.25 network.

In some cases a company may already have coaxial cable installed to support 3270 terminals attached to existing cluster control units such as the IBM 3741. In this case the DFT link protocol can be used to connect clients to Host Integration Server 2000 over these cables. Now that most companies have converted from coaxial cables to LANs, however, this method of attachment is becoming more rare.

For Server to AS/400 access, Host Integration Server 2000 supports the same protocols discussed above. Instead of DFT, however, Host Integration Server 2000 supports the Twinax coaxial cable connection method that was originally used to connect IBM 5250 terminals directly to an AS/400.

Client to server protocols

In order to allow Host Integration Server 2000 clients to communicate with Host Integration Server 2000 Servers, Host Integration Server 2000 supports the following protocols:

- TCP/IP
- IPX/SPX
- Microsoft Networking (named pipes)
- AppleTalk

We discussed TCP/IP and DLC 802.2 above. IPX/SPX is used primarily in a NetWare network, while AppleTalk is used to connect Macintosh clients to Host Integration Server 2000.

Server-to-server protocols

In addition to the server-to-host and client-to-server communications discussed above, Host Integration Server 2000 servers also need to communicate directly with each other. They need to do this in order to exchange information on network resource changes as well as to support other inter-server features of Host Integration Server 2000 such as the Distributed Link Service, Downstream connections support and PU passthrough. Host Integration Server 2000 supports a subset of the Host Integration Server 2000 Client/Server protocols for this purpose. Normally Host Integration Server 2000 servers will use TCP/IP for this. IPX/SPX is available for use in a NetWare based network for backward compatibility when Host Integration Server is running under Windows NT.

Terminal Emulation Services

In addition to providing network protocols, the Network layer also includes terminal emulation services.

Starting in the early 1970s IBM produced the 3270 family of display stations, printers and terminal cluster control units. Terminal devices such as displays and printers were normally connected to a cluster control unit via coaxial cables, which were then connected directly to the mainframe of to a front-end processor. Support for these devices came via the LU type 2 support that

we discussed earlier in this paper. Somewhat later IBM also introduced the AS/400 mid-range computer system. These systems supported the IBM 5250 family of terminals. The 5250 fulfilled the same role as the 3270 in mainframe applications. Instead of using LU 2, however, it used LU 6.2 as a communications protocol.

Over time these relatively unintelligent terminals were replaced with personal computers. To continue to run the same applications, special client-based software was developed to emulate the functions of the original 3270 and 5250 terminals. IBM and other companies subsequently provided client-based terminal emulators such as Attachmate Extra! and Rumba. These client-based terminal emulators run completely on the client computer performing all protocol conversion and display/printer services locally.

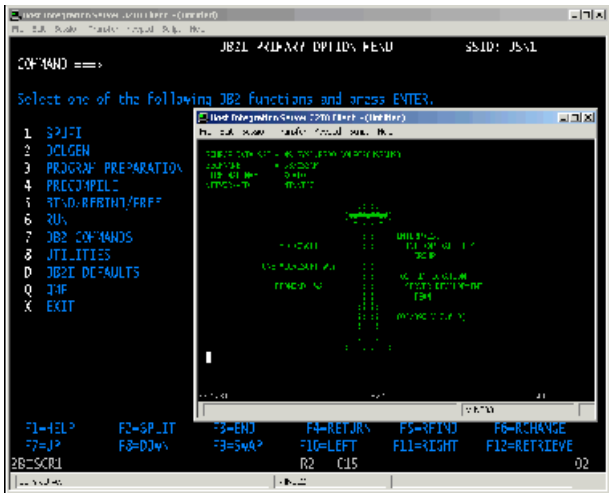


Figure 13. 3270 Emulation Sessions. Host Integration Server 2000 offers a simple, multi-session, 3270 Client, while supporting popular third-party 3270 client programs.

Other emulators however, make use of a gateway approach to put part of the protocol stack on the client and part on a server. Host Integration Server 2000, and emulators that work with it use this approach. This splits the protocol stack and allows the emulator to have a smaller footprint on the client. Host Integration Server 2000 comes with basic 3270 and 5250 emulators that can be used to verify the correct installation and configuration of Host Integration Server 2000. These emulators are generally not suitable for production use, so one of the third party products needs to be acquired for this purpose. The Host Integration Server 2000 emulators also support file transfer and the operation of multiple terminal sessions on one desktop

The Host Integration Server 2000 emulators (and most third party emulators as well) also support the ability to automate mainframe sign on using a scripting language. This allows the emulator to work with the Host Integration Server 2000 *Mainframe Security Integration* features such as the Single Sign On feature discussed in the Management section of this paper.

To allow for even lighter weight clients, Host Integration Server 2000 also supports a version of 3270 and 5250 emulation that operates directly over TCP/IP. Since TCP/IP is the only client-side requirement and it normally already exists on the client, there is no need for any part of the SNA stack of Host Integration Server 2000 client to be located on the workstation. *TN3270E and TN5250* allow a user to access the host computer TCP/IP network using a variation of the TCP/IP Telnet protocol.

Most 3270 emulators also support the ability to transfer files between a host computer and a workstation using the IND$FILE utility program. This program works in conjunction with a host operating system such as TSO or transaction processing monitor software such as CICS running on the host. It allows the client to request a file transfer manually and monitor the results.

In addition, Host Integration Server 2000 networking provides support for most of the 3270 and 5250 printing protocols such as those supported by LU types 1, 3 and 6.2 (APPC) devices.

Host Integration Server 2000 extends and enhances the already dominant support of network interconnections provided by prior versions of SNA Server.

Conclusion

Host Integration Server 2000, represents the latest evolutionary step on the way to a complete mainframe and peer host interoperability solution. Moving forward, Microsoft will focus on providing bi-directional interoperability with AS/400 and IBM mainframes computers at key interoperability layers. Additionally, Microsoft will work with third parties to provide add-on products to Host Integration Server 2000 that enhance cross-platform interoperability. As you can see, Host Integration Server 2000 is a product with which you can build your Web solutions today, while using it as a solid platform to support Microsoft .NET tomorrow.

Microsoft MSMQ-MQSeries Bridge Performance Results

Host Integration Server 2000
Microsoft Corporation

October 2001

Summary: Microsoft Host Integration Server 2000 (HIS) includes a number of components that enable integration with applications on IBM mainframe and AS/400 minicomputers as well as on other platforms. The Microsoft MSMQ-MQSeries Bridge included with Host Integration Server 2000 and SNA Server 4.0 provides a gateway between IBM MQSeries messaging applications predominant on IBM host computers and Microsoft Message Queuing (MSMQ) used on Windows systems. This article reports on performance results based on testing the MSMQ-MQSeries Bridge supplied with Host Integration Server 2000 and SNA Server 4.0. (13 printed pages)

Contents

Introduction

Microsoft® Host Integration Server 2000 includes a set of *Application Integration* components, which provide desktop or server-based applications with access to host applications. The *Application Integration* components included in Host Integration Server 2000 include the following:

- Microsoft MSMQ-MQSeries Bridge
- COM Transaction Integrator (COMTI) for CICS and IMS

The Microsoft MSMQ-MQSeries Bridge included with Host Integration Server 2000 and SNA Server 4.0 provides a gateway between IBM MQSeries messaging applications predominant on IBM host computers and Microsoft Message Queuing (MSMQ) used on Windows systems.

This article reports the results of performance testing using the Microsoft MSMQ-MQSeries Bridge provided with Host Integration Server 2000 and SNA Server 4.0 with Service Pack 3.

Based on testing on a dual processor Pentium II 400 MHz computer, the MSMQ-MQSeries Bridge is capable of sustaining 450 transactions per second (TPS) for one kilobyte non-transactional round-trip messages. This represents a total sustained message throughput of 900 messages per second.

Based on testing on a quad processor Pentium II Xeon 400 MHz computer, the MSMQ-MQSeries Bridge is capable of sustaining 900 transactions per second (TPS) for one kilobyte non-transactional round-trip messages. This represents a total sustained message throughput of 1,500 messages per second.

The MSMQ-MQSeries Bridge is not CPU bound, which allows other more processor-intensive applications to run on the same computer.

The following tables show a quick summary of the results based on performance testing the MSMQ-MQSeries Bridge with a distributed version of the Component Object Model (DCOM) client load.

Table 1. Non-transactional messages throughput using MSMQ-MQSeries Bridge (messages/second)

MSMQ-MQSeries Bridge Test Description	SNA Server 4.0 Service Pack 3	HIS 2000 1 MSMQ Queue Manager	HIS 2000 3 MSMQ Queue Managers
MSMQ to MQSeries 1 KB roundtrip messages (1 KB send/1 KB receive)	740	740	1050 (Note 2)
MSMQ to MQSeries 1 KB one-way messages (1 KB send to MQSeries queue)	425	450	975 (Notes 2)
MSMQ to MQSeries (1 KB send/8 KB receive messages)	350	450	600
MQSeries to MSMQ 1 KB one-way messages (1 KB send to MSMQ queue)	370	370	1010
MQSeries to MSMQ 8 KB one-way messages (8 KB send to MSMQ queue)	250	345	495

Notes:

1. All messages contained character data only.
2. MSMQ was the factor limiting performance in this configuration.

Table 2. Transactional messages throughput using MSMQ-MQSeries Bridge (messages/second)

MSMQ-MQSeries Bridge Test Description	SNA Server 4.0 Service Pack 3	HIS 2000 1 MSMQ Queue Manager
MSMQ to MQSeries 1 KB one-way messages (1 KB send only)	130	130
MSMQ to MQSeries 1 KB round-trip messages (1 KB send/1 KB receive)	75	75
MQSeries to MSMQ 1 KB round-trip messages (1 KB send/1 KB receive)	197	197
MQSeries to MSMQ 8 KB round-trip messages (8 KB send/8 KB receive)	85	165

Notes:

1. All messages contained character data only.

Test Methodology and Environment

The goal of testing was to simulate a typical corporate messaging network and examine the behavior of the MSMQ-MQSeries Bridge as it was subjected to an ever-increasing message load. Testing was done at the Microsoft Enterprise Interoperability Group Performance Laboratory at corporate headquarters in Redmond, Washington during June 2000.

The Test Methodology

The test methodology for this comparison centered on a simulation of interactive transaction processing to deliver messages to the Microsoft MSMQ and IBM MQ Series networks. Using the Microsoft MSMQ-MQSeries Bridge, MSMQ, and IBM MQSeries, applications can send messages to each other through the message queuing systems. The MSMQ-MQSeries Bridge achieves this by mapping the messages and data fields of the sending system and the values associated with those fields to the fields and values of the receiving environment. After the mapping and conversion, the MSMQ-MQSeries Bridge completes the process by routing the message across the combined MSMQ and MQSeries networks.

A Microsoft test tool was used to create simulated messages. The tool, designed to provide basic stress testing for the MSMQ-MQSeries Bridge, works by sending or receiving messages from a predetermined queue. The tool is configurable to create any number, size, and type of message that is required to test the MSMQ-MQSeries Bridge features. The test software was written in Microsoft® Visual Basic® using Microsoft ActiveX® and DCOM. Separate control center software launched the MSMQ and MQSeries clients on the client computers and controlled the type and number of messages sent by the client each second. By gradually increasing the number of messages sent by the client computers, it was possible to determine the maximum values for sustained throughput that could be maintained by the MSMQ-MQSeries Bridge.

Separate tests were conducted of the MSMQ-MQSeries Bridge included in Host Integration Server 2000 running on Microsoft® Windows® 2000 Advanced Server and the MSMQ-MQSeries Bridge included with SNA Server 4.0 SP3 running on Microsoft Windows NT® 4.0 Enterprise Server. In both series of tests, similar hardware was used by the computer running the MSMQ-MQSeries Bridge software.

The MSMQ or MQSeries client workstations each went through cycles of sending/receiving a specified rate of messages per second set by the control center application. This request frequency and the number of client workstations used generated the resulting total messages per second load on the MSMQ or MQSeries servers and the MSMQ-MQSeries Bridge. The message and transaction load was increased incrementally, and test data was collected after each increase in client load was added to the test bed configuration. Client loads were increased until the messaging Connector Queue backed up and could not flush itself within a reasonable amount of time.

Key Testing Metrics

The test methodology was based on using a number of key metrics for determining performance. In order to measure some of these metrics, a separate computer running the Microsoft Network Monitor software (NetMon, a protocol analyzer) was placed on each network segment. An analysis of the NetMon protocol analysis logs was used to determine the recorded data transaction throughput and total LAN traffic loads.

The MQSeries libraries and messaging APIs do not have a way to check arrival time in the MQSeries queue. Since dequeueing from an MQSeries queue is slower than enqueueing, there needs to be a way to see how quickly the queue is populated. A separate MQSeries ActiveX DLL downloaded from the IBM Web site was used for this purpose. Performance counters on CPU usage, disk writes, and other systems measures were retrieved via the Perfmon application.

The key testing metrics include the following:

- **Transactions per second (TPS)**—the number of transactions the MSMQ-MQSeries Bridge was able to send and receive each second. Each transaction consisted of one request message and one corresponding reply message. This test metric was determined using NetMon.
- **Messages per second (msg/sec)**—the number of messages the MSMQ-MQSeries Bridge was able to send and receive each second. Each message could be a request message or a reply message. For bi-directional messages of the same size, the value for transactions per second represented 50% of the value for messages per second. This test metric was determined using NetMon.
- **CPU Load**—the CPU utilization on the computer system running the MSMQ-MQSeries Bridge application. This test metric was determined using performance counters gathered by Perfmon.

The Test Environment

The test environment consisted of two private network segments running fast Ethernet (100Base-T): one for MSMQ and one for MQSeries. The computer running the MSMQ-MQSeries Bridge contained two fast Ethernet network interface cards and connected these two segments. All testing was based on using TCP/IP as the network protocol for connecting to both the MSMQ and MQSeries segments.

The MSMQ network segment included the following:

- Multiple MSMQ client workstations to create the MSMQ message load.
- A server computer functioning as a domain controller running MSMQ server software.
- A test computer running NetMon.

When testing the MSMQ-MQSeries Bridge included with SNA Server 4.0 on Windows NT 4.0, the domain controller on the MSMQ segment also functioned as the MSMQ Primary Enterprise Controller (PEC).

The MQSeries network segment included the following:

- Multiple MQSeries client workstations to create the MQSeries message load.
- Multiple server computers running MQSeries server software.
- A test computer running NetMon.

When testing the MSMQ-MQSeries Bridge included with SNA Server 4.0 on Windows NT 4.0, multiple computers were functioning as MQSeries servers. On Windows 2000, a later version of MQSeries software was used, and only a single computer acted as the MQSeries server.

The following figure depicts the network topology used to test the MSMQ-MQSeries Bridge included in Host Integration Server 2000. Note that the computer operating as the network monitor running NetMon is not shown in this figure.

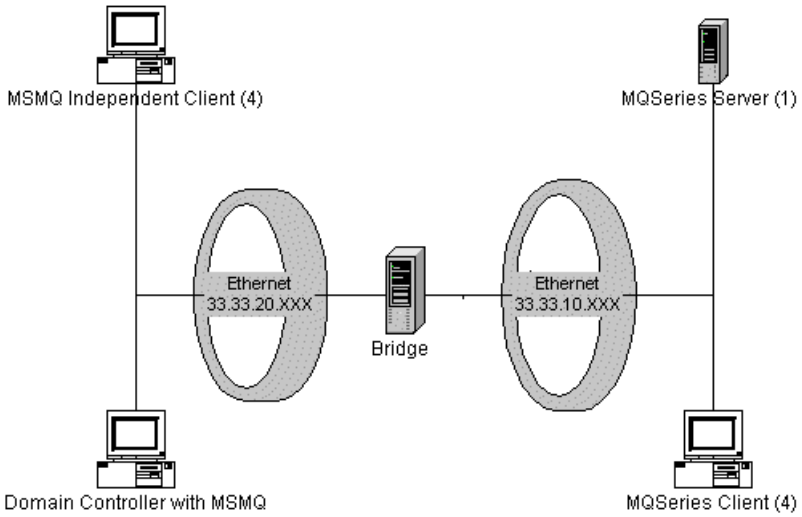


Figure 1. Network topology for testing MSMQ-MQSeries Bridge on Host Integration Server 2000

Hardware platforms for Host Integration Server 2000 tests

The following hardware was used for testing the MSMQ-MQSeries Bridge included with Host Integration Server 2000.

MSMQ-MQSeries Bridge Machine: A dual processor Pentium II 400 MHz computer with 512 MB of RAM was configured to run Microsoft Windows 2000 Advanced Server, IBM MQSeries Client v5.1, Microsoft MSMQ v2.0 with routing enabled, and the Microsoft MSMQ-MQSeries Bridge included with Host Integration Server 2000.

MQSeries Client Machines: Pentium II 350 MHz computers with 128 MB of RAM were configured to run Microsoft Windows 2000 Professional and MQSeries Client v5.1.

MQSeries Server Machine: An eight-processor Pentium III 550 MHz computer with 4 GB of RAM was configured to run Microsoft Windows 2000 DataCenter and IBM MQSeries Server v5.1.

MSMQ DC Machine: A dual processor Pentium II 400 MHz computer with 512 MB of RAM was configured to run Microsoft Windows 2000 Advanced Server as the domain controller and Microsoft MSMQ v2.0.

MSMQ Client Machines: Pentium II 350 MHz computers with 128 MB of RAM were configured to run Microsoft Windows 2000 Professional and MSMQ v2.0.

The following table describes in more detail the specific hardware used for testing the MSMQ-MQSeries Bridge included in Host Integration Server 2000.

Table 3. Specific hardware included in Host Integration Server 2000

Function	Vendor	Operating System	Processor	RAM	Network Adaptor
MSMQ Clients (4)	Ciara	Windows 2000 Professional	PII 350 MHz	128 MB	Intel Ether Express Pro
MQSeries Clients (4)	Ciara	Windows 2000 Professional	PII 350 MHz	128 MB	Intel Ether Express Pro
Bridge	Ciara	Windows 2000 Advanced Server	Dual PII 400 MHz	512 MB	Intel Ether Express Pro
Domain Controller and MSMQ Server	Ciara	Windows 2000 Advanced Server	Dual PII 400 MHz	512 MB	Intel Ether Express Pro
MQSeries Server	Fujitsu	Windows 2000 Datacenter	Eight-processor PIII 550 MHz	4 GB	Intel Pro /100+ Server

The following figure depicts the network topology used to test the MSMQ-MQSeries Bridge included in SNA Server 4.0 Service Pack 3. Note that the computer operating as the network monitor running NetMon is not shown in this figure.

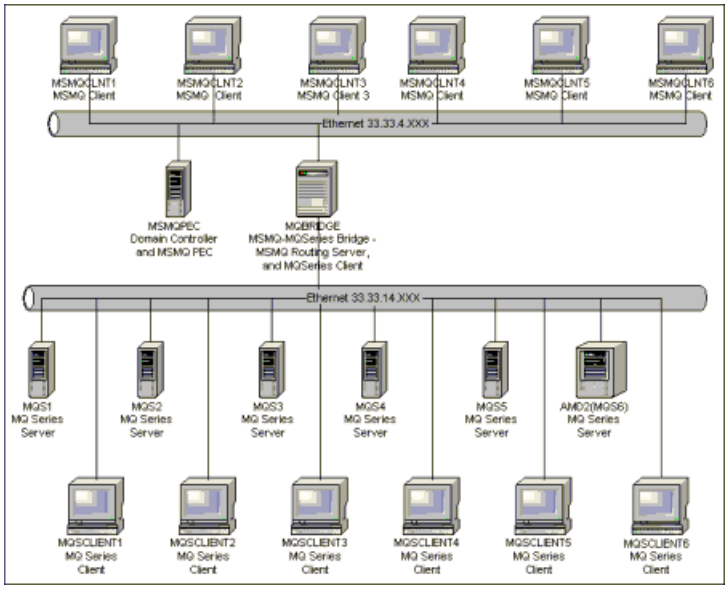


Figure 2. Network topology for testing MSMQ-MQSeries Bridge on SNA Server 4.0 (click image to see larger picture)

Hardware platforms for SNA Server 4.0 tests

The following hardware was used for testing the MSMQ-MQSeries Bridge included with SNA Server 4.0 with Service Pack 3.

MSMQ-MQSeries Bridge Machine: A dual processor Pentium II 400 MHz computer with 512 MB of RAM was configured to run Microsoft Windows NT 4.0 Enterprise Server with Service Pack 5, IBM MQSeries Client v5.0, Microsoft MSMQ v1.0 Routing Server, and the Microsoft MSMQ-MQSeries Bridge included with SNA Server 4.0 SP3.

MQSeries Client Machines: Pentium II 350 MHz computers with 128 MB of RAM were configured to run Microsoft Windows NT 4.0 Workstation and MQSeries Client v5.0.

MQSeries Server Machines: Five dual processor Pentium II 400 MHz computers with 512 MB of RAM were configured to run Microsoft Windows NT 4.0 Server with Service Pack 5 and IBM MQSeries Server v5.0. One additional quad processor Pentium Pro 200 MHz computer with 512 MB RAM was configured to run Microsoft Windows NT 4.0 Server with Service Pack 5 and IBM MQSeries Server v5.0.

MSMQ DC Machine: A dual processor Pentium II 400 MHz computer with 512 MB of RAM was configured to run Microsoft Windows NT 4.0 Server with Service Pack 5 as the domain controller, Microsoft® SQL Server™ 7.0, and Microsoft MSMQ v1.0 as the MSMQ Primary Enterprise Controller (PEC).

MSMQ Client Machines: Pentium II 350 MHz computers with 128 MB of RAM were configured to run Windows NT 4.0 Workstation and MSMQ v1.0.

The following table describes in more detail the specific hardware used for testing the MSMQ-MQSeries Bridge included in SNA Server 4.0 with Service Pack 3.

Table 4. Specific hardware included in SNA Server 4.0 with SP3

Function	Vendor Model	Operating System	Processor	RAM	Network Adaptor
MSMQ Clients (2)	Dell Optiplex GX1	Windows NT Workstation with SP5	PII 350 MHz	128 MB	3Com Fast Etherlink XL
MSMQ Clients (4)	Ciara	Windows NT Workstation with SP5	PII 350 MHz	128 MB	Intel Ether Express Pro
MQSeries Clients (6)	Ciara	Windows NT Workstation with SP5	PII 350 MHz	128 MB	Intel Ether Express Pro
Bridge	Dell Precision 610	Windows NT 4.0 Enterprise Server with SP5	Dual PII 400 MHz	512 MB	Intel Ether Express Pro
Domain Controller and MSMQ Server	Ciara	Windows NT 4.0 Enterprise Server with SP5	Dual PII 400 MHz	512 MB	Intel Ether Express Pro
MQSeries Servers (5)	Ciara	Windows NT 4.0 Server with SP5	Dual PII 400 MHz	512 MB	Intel Ether Express Pro
MQSeries Server (1)	Amdahl Envista	Windows NT 4.0 Server with SP5	Quad Pentium Pro 200 MHz	512 MB	Intel Ether Express Pro

Performance Settings

The MSMQ-MQSeries Bridge has several definable attributes that can affect performance and message throughput. These settings can be viewed and changed using the MSMQ-MQSeries Bridge Explorer by selecting a Bridge instance and right clicking on properties.

The MSMQ-MQSeries Bridge software creates and uses four message pipes for message transport as follows:

- MSMQ to MQSeries messages sent with normal service (transactional)
- MSMQ to MQSeries messages sent with high service (non-transactional)
- MQSeries to MSMQ messages sent with normal service (transactional)
- MQSeries to MSMQ messages sent with high service (non-transactional)

An important setting on the **Advanced** tab is the number of threads that are used by the MSMQ-MQSeries Bridge software to service each of these message pipes. Ideally, this value would be set to reflect the number of MQSeries and MSMQ queue managers that the Bridge will service based on the number of CPUs. However, some limited testing was done using varying numbers of threads. The results indicated that there is no noticeable difference in performance if more threads are allocated than the number of MQSeries Queue Managers being serviced. However, if the thread count is lower than the number of MQSeries Queue Managers that are to be serviced, the Bridge performance drops depending on the number of messages sent and the number of threads allocated. This results from the number of jobs servicing the Queue Managers competing for access to the available threads.

Other settings that have an impact on MSMQ-MQSeries Bridge are batch attributes on each individual message pipe. A batch is a group of messages that get processed by the Bridge at the same time. These settings can be viewed and changed using the MSMQ-MQSeries Bridge Explorer by selecting a Bridge instance and right clicking on the properties for each of the four message pipes. The **Batch** tab exposes three properties on a message pipe that affect the number and size of batches used for each message pipe.

Table 5. Batch properties

Batch Property	Comments
Max. Number of Messages	The maximum number of messages in a batch (defaults to 10).
Max. Accumulated Size	The maximum size in bytes of a batch (defaults to 1024 bytes).
Max. Accumulated Time	The maximum time in milliseconds during which messages are batched (defaults to 512).

Transmission begins as soon as there are messages to be sent. When any of the above limits is reached, the message pipe checks that the batch was fully received on the destination side.

To improve Bridge performance, these batch properties were set as follows during testing:

- Max. Number of Messages: 1,000
- Max. Accumulated Size: 300,000
- Max. Accumulated Time: 256

Test Results

Test Results for Round-Trip 1-KB Messages

Non-transactional

A single MQSeries Queue Manager through a single MSMQ-MQSeries Bridge using the SNA Server 4.0 SP3 version provided a maximum sustained rate of 370 TPS for a total of 740 msg/sec through the MSMQ-MQSeries Bridge. When the same test was run using the MSMQ-MQSeries Bridge in Host Integration Server 2000, the sustained rate stayed the same at 370 transactions per second.

By increasing the number of MQSeries Queue Managers that the MSMQ-MQSeries Bridge serviced to three (or tripling the number of active message pipes), the rate increased to 525 TPS for a total of 1050 msg/sec. The limiting factor was the number of outgoing messages per second that MSMQ could sustain to the connector queues of MSMQ routing server (350 msg/sec).

Transactional

The sustained demand for a single MQSeries Queue Manager through a single MSMQ-MQSeries Bridge was 65 TPS for a total of 130 msg/sec through the MSMQ-MQSeries Bridge in both SNA 4.0 SP3 and Host Integration Server 2000. The MSMQ-MQSeries

Bridge CPU load was less than 10%, but the disk queue length was 1.3. A disk queue length of 2.0 represents the maximum disk processing possible. The test results indicate that the disk was being heavily used, but still had 35% of the disk processing idle.

Test Results for Round-Trip MSMQ 1-KB MQSeries 8-KB Messages

Non-transactional

While testing on the MSMQ-MQSeries Bridge using a single MSMQ message connector pipe and one MQSeries message connector pipe, the maximum sustained rate was 175 TPS for a total of 350 msg/sec using the MSMQ-MQSeries Bridge in SNA Server 4.0 SP3. The maximum sustained rate increased to 225 TPS for a total of 450 msg/sec when using the MSMQ-MQSeries Bridge in Host Integration Server 2000.

Adding two additional MQSeries Queue Managers to the Host Integration Server 2000 version of the MSMQ-MQSeries Bridge (total of 3 Queue Managers) increased the maximum sustained rate to 300TPS or 600 msg/sec.

Test Results for MQSeries to MSMQ One Way 1 KB Messages

Non-Transactional

On the MSMQ-MQSeries Bridge in SNA Server 4.0 SP3, the maximum sustained rate was 370 msg/sec. The messages were sent from one Queue Manager to one independent client. The average CPU usage for the MSMQ-MQSeries Bridge was 19 percent. Disk input/output and memory usage was relatively minor. Using the MSMQ-MQSeries Bridge in Host Integration Server 2000 had no impact on this maximum sustained rate. On Host Integration Server 2000, the processor, disk, and memory usage were virtually the same as the results on SNA Server 4.0 SP3.

When two additional MQSeries Queue Managers were added to the Host Integration Server 2000 MSMQ-MQSeries Bridge (total of 3 Queue Managers), the maximum sustained rate handled was 1010 msg/sec. CPU load increased to 57 percent, but the memory and disk usage remained virtually the same.

Transactional

Transactional messages guarantee delivery, delivering once and only one message. Changing to transactional messages decreases the total throughput, as would be expected. Using the MSMQ-MQSeries Bridge in SNA Server 4.0 SP3, the maximum sustained rate handled was 197 msg/sec. The CPU load was similar at 20 percent, but disk activity increased to 0.95 of the queue length. A disk queue length of 2.00 indicates a particular disk queue is saturated.

Testing the MSMQ-MQSeries Bridge in Host Integration Server 2000 did not make a difference in the throughput. The maximum sustained rate handled was 197 msg/sec. CPU load, however, dropped to 15 percent. Disk activity was unchanged with a queue length at 0.95.

Test Results for MQSeries to MSMQ One Way 8 KB Messages

Non-transactional

Increasing the message size had the expected effect. The maximum sustained rate that the MSMQ-MQSeries Bridge in SNA Server 4.0 SP3 could handle was 250 msg/sec. The MSMQ-MQSeries Bridge CPU usage stayed consistent only increasing to 21 percent. Disk activity increased to 0.25 queue length. The maximum sustained rate increased to 345 msg/sec. when changing the MSMQ-MQSeries Bridge in Host Integration Server 2000. Disk activity stayed at 0.25 queue length.

Adding two additional MQSeries Queue Managers to the MSMQ-MQSeries Bridge in Host Integration Server 2000 (a total of 3 Queue Managers) increased the sustained throughput to 495 msg/sec. The MSMQ-MQSeries Bridge CPU load increased to almost 70 percent and disk queue length was 0.87.

Transactional

Changing to transactional messages and increasing the message size slows the throughput as would be expected. For the MSMQ-MQSeries Bridge in SNA Server 4.0 SP3, the maximum sustained throughput was 85 msg/sec. The CPU load, however, decreased to about 10 percent and the disk queue length was at 0.56. Using the MSMQ-MQSeries Bridge in the Host Integration Server 2000 increased the maximum sustained throughput to 165 msg/sec. The performance counters in both MSMQ-MQSeries Bridge computers were the same with 10 percent CPU load and 0.54 disk queue length.

Test Results for MSMQ to MQSeries One Way 1 KB Messages

Non-transactional

The MSMQ-MQSeries Bridge in SNA Server 4.0 SP3 was able to sustain a rate of 425 msg/sec with one message pipe from MSMQ to MQSeries without any buffering. The messages were sent from one independent client to one Queue Manager. The MSMQ-MQSeries Bridge CPU usage averaged to 34 percent over the 1000-second test. Disk input/output was low (queue length was 0.03). Using the MSMQ-MQSeries Bridge in Host Integration Server 2000, the throughput increased to 450 msg/sec. The CPU load average was again 34 percent with a disk queue length 0.03.

Boosting the number of MQSeries Queue Managers from one to three, the expected outcome would be an increase in maximum sustained throughput. Tests on the MSMQ-MQSeries Bridge in Host Integration Server 2000 indicate this to a certain extent. The MSMQ-MQSeries Bridge using three MQSeries Queue Managers was able to sustain a rate of 975 msg/sec with a CPU load at 84 percent and disk queue length at 0.086. However, the throughput could be conceivably higher since the limiting factor was the sustained outgoing messages per second from the MSMQ client to the MSMQ routing server connector queue (about 325 msg/sec per client).

Transactional

Testing with transactional messages, the sustained rate through the MSMQ-MQSeries Bridge in a SNA Server 4.0 SP3 averaged 75 msg/sec. At this rate the CPU load was only running at 8 percent, but the disk queue length averaged to 1.25. Using the MSMQ-MQSeries Bridge in Host Integration Server 2000, the maximum sustained demand remained at 75 msg/sec. The CPU load was reduced to 4 percent and the disk queue length was decreased to 1.01. The messages began to back up on the MSMQ client machine before they started to back up on the MSMQ-MQSeries Bridge thus causing the test to stop at the 75 msg/sec rate.

Observations

MSMQ transactional messages back up on the client side after 75 to 100 messages per second. This causes the transactional message tests to stop and show low messages per second (below 10 msg/sec).

MSMQ non-transactional messages back up on the client side above 425 messages per second when sending to a single MQSeries Queue Manager. When sending to multiple MQSeries queue managers, the average rate drops to about 350 messages a second.

Roughly every 109 milliseconds, the MSMQ-MQSeries Bridge sends out a 64-byte message to each of the active Connector Queues on MQSeries to determine if there are any messages. This polling results in a minimum of ten 64-byte messages per second to each active MQSeries message pipe. This is adjustable via a registry setting in Host Integration Server 2000.

MQSeries takes 150-200 milliseconds to close a queue and 15-16 milliseconds to open a queue. This is an expensive operation, so leaving a queue open for long periods of time proves more advantageous for performance.

With non-transactional messages, if the queue depth on the client or any message pipe to a connector queue grows above 32,000 1-KB messages, the MSMQ service (mqsvc.exe) starts to take processor time away from all other services. The processor time is not spent on sending the messages, as the outgoing messages per second rate decreases to 100-200 messages per second.

If a variant full of strings is put into the body of a MSMQ message, the message then gets converted to Unicode, thus doubling the size of the message. However, the MSMQ-MQSeries Bridge and MQSeries recognize the message as ANSI format. For example, one KB of text converts to 2 KB when placed into a MSMQ message body. The MSMQ-MQSeries Bridge defines the size of the message as 1 KB and MQSeries defines the body size as 1 KB.

The MSMQ Visual Basic plug-in does not allow you to specify the type of message body created. The default is string, which is then converted to Unicode. The MSMQ C/C++ API allows the message type to be specified, so these interfaces should support increased performance since the MSMQ message body could be decreased by 50 percent for text messages of strings. The Visual Basic plug-in for the MQSeries client allows the format of the message body to be specified.

If a message is sent from MQSeries to MSMQ, the only prerequisite is that the Remote Queue Manager is identified in MQSeries Queue Manager. If a message is sent to a non-existent queue, the overall performance of the MSMQ-MQSeries Bridge is decreased and the messages wind up in the MSMQ-MQSeries Bridge Dead Letter Queue. MQSeries is not notified that the message could not be delivered. On the other hand, MSMQ will not allow you to even open the queue if it is not identified in the MSMQ GUI. If it is not identified on the MQSeries Queue Manager, the message is delivered to the MSMQ-MQSeries Bridge Dead Letter queue. Transactional messages stay in the outgoing queue until the message expires. This also affects the performance through the MSMQ-MQSeries Bridge causing it to decrease to 40 messages per second when there are only a few messages in the Dead Letter queue. Performance can decrease to less than 1 message per second if there are tens of thousands of messages in the outgoing queue. Performance on the MSMQ-MQSeries Bridge does degrade while these messages are passed through the MSMQ-MQSeries Bridge.

If a quota limit is not set for the maximum number of messages resident in the MSMQ service and messages begin to pile up in the queue, then performance of the MSMQ-MQSeries Bridge begins to degrade. Memory is associated with each message in the MSMQ service, so the Available Bytes of Memory on the MSMQ-MQSeries Bridge computer is reduced. If enough messages are put in to the queue, the machine becomes sluggish and unresponsive.

Final Thoughts

In regard to the limitations of the hardware used for testing and the messaging software used to put messages into the MSMQ-MQSeries Bridge, it appears the MSMQ-MQSeries Bridge is, as a whole, virtually transparent in sending messages between MSMQ and MQSeries. There are optional fields (**Reply To Queue Manager**, for example) that, when populated, increase the amount of time required to send messages through the MSMQ-MQSeries Bridge. As new features (encryption, for example) are added to the MSMQ-MQSeries Bridge, the amount of time to send messages will increase, thus causing the number of messages per second to decrease and the MSMQ-MQSeries Bridge process to use more CPU. Limiting the number of times that the MSMQ protocol and MQSeries protocol are needed by the MSMQ-MQSeries Bridge will be instrumental in keeping the amount of overhead observed by the MSMQ-MQSeries Bridge to a minimal level.

Both MSMQ and MQSeries messaging software make it expensive on the network to open and close queues. However, if a queue is not closed after long periods of time, it may have detrimental consequences on the machine. The performance testing software opened the queue and sent all messages in the specified time (5000 messages in 10 seconds, for example), and then closed the queue. Previous testing had determined that opening and closing the queue for each message caused the TPS to decrease a minimum of 20 percent.

In comparing the MSMQ-MQSeries Bridge on the Windows NT 4.0 and Windows 2000 platforms, there is one noticeable difference. MSMQ seems to have degraded its performance in the Connector queue. With MSMQ v1.0 and MQSeries v5.0, the limiting factor was that MQSeries was bottlenecked at 500+ messages. Now with MSMQ v2.0 and MQSeries v5.1, the maximum sustained rate that can be pumped into the Connector Queue from MSMQ is about 425 messages a second. An internal test of MQSeries put the enqueueing at over 600 messages per second and the dequeueing at over 575 messages per second on an eight-processor machine. The disk activity, processor activity, and memory usage were very low (less than 10 percent processor time, 0.2 disk queue length, and 5 percent memory usage).

On the other hand, the changes made to MSMQ and MQSeries have increased the performance of the queues. On the Windows NT 4.0 platform, the one-way MSMQ to MQSeries messages measured at 500 messages per second and MQSeries to MSMQ measured 565 messages per second sustained. However, adding both pipes brought the total sustained messages per second down to 600 (or 300 messages from either side) messages per second. Now, on the surface, the number of messages sent has decreased in a one-direction fashion, but increased on multiple pipes being used. The MSMQ-MQSeries Bridge did not cause this one directional slow down. Internal testing on MSMQ determined that a sustained rate of 400 messages per second is expected.

Sample Programs for COMTI

Host Integration Server 2000
Microsoft Corporation

October 2001

Summary: This article describes the sample programs for application integration using COMTI (COM Transaction Integrator) for CICS and IMS included with the Host Integration Server 2000 Software Development Kit. (4 printed pages)

Contents

Introduction

The source code for several sample programs that illustrate using features of the Microsoft® COM Transaction Integrator (COMTI) for CICS and IMS is included on the Microsoft® Host Integration Server 2000 CD-ROM as a part of the Host Integration Server Software Development Kit (SDK). COMTI allows developers to integrate component-based Microsoft Windows® applications using COM, distributed COM, and COM+ with CICS and IMS transactions on IBM mainframes.

In addition to the COMTI samples included in the Host Integration Server SDK, there is a basic COMTI sample titled CedarBank that is installed with COMTI when the COMTI feature option is selected during setup. The CedarBank sample is installed under the `system\Tutorials\CedarBank` subdirectory below where Host Integration Server is installed (the default location is `C:\Program Files\Host Integration Server\system\Tutorials\CedarBank`).

Note that documentation on COMTI is not included with the Host Integration Server SDK. Documentation on COMTI is included under Application Integration Services as part of the Host Integration Server 2000 user documentation. The documentation is also available in printable format on the Host Integration Server CD-ROM under the `Documentation\Printable Books` folder in the `Application Integration Services.pdf` file.

The COMTI sample programs are located in the `\SDK\Samples\COMTI` subdirectory on the Host Integration Server 2000 CD-ROM. These files are copied to your hard drive during Host Integration Server software or Host Integration Client software installation when the Host Integration Server Software Development Kit option is selected. These samples are installed in the `Samples\COMTI` subdirectory below where the Host Integration Server SDK software is installed (`C:\Program Files\Host Integration Server SDK`, by default).

When installed as part of the MSDN Platform SDK, these samples are located under the `Samples\NetDS\HIS\COMTI` subdirectory below where the MSDN Platform SDK has been installed (`C:\Program Files\Microsoft SDK`, by default).

These sample programs include the files in the following subdirectories:

Subdirectory	Description
BoundedRecordsets\COBOL-CICS	A sample program in COBOL using COMTI that illustrates the use of b ounded recordsets. This subdirectory also contains a sample TLB file cr eated using the COMTI Component Builder for this COBOL sample.
BoundedRecordsets\VB	A sample class defined in Microsoft® Visual Basic® using COMTI that illustrates the use of bounded recordsets.
ProgrammingSpecifics	This folder contains a comprehensive sample that includes Visual Basic client code as well as mainframe COBOL code and sample COMTI type libraries. This sample is intended to be a complete end-to-end sample demonstrating features of COMTI.
ProgrammingSpecifics\CICSNonlink	A sample program in COBOL using COMTI that demonstrates how to r eceive a COMTI fixed-sized data area greater than 32767. This sample i s not intended to be a complete end-to-end program, but it demonstra tes the receiving-side logic of a CICS Non-Link server application prog ram using COMTI.
ProgrammingSpecifics\TCP	A set of several sample programs in COBOL using COMTI that demons trate how to use a CICS TCP server application.

SyncLev2	A sample program in COBOL using COMTI that demonstrates how to use Sync Level 2.

These samples primarily use a remote environment of CICS using LU 6.2. These COMTI samples are designed to assist developers in creating code for specific COMTI features.

In order to first start working with COMTI, it is recommended that developers use the CedarBank tutorial that comes with the COMTI installation. The CedarBank tutorial illustrates how to use all of the COMTI Remote Environments and includes the COMTI type libraries and the COBOL code for the mainframe for all of the environments (IMS, CICS, APPC, and TCP/IP). The CedarBank sample also includes sample programs for the client-side code written in Microsoft Visual Basic and Microsoft Visual C++®.

Once connectivity has been established by working with the CedarBank tutorial, then the COMTI samples included with the Host Integration Server SDK can be used to gain an understanding of more advanced COMTI features not covered by the CedarBank tutorial.

Bounded Recordsets Sample

The Microsoft® COM Transaction Integrator (COMTI) for CICS and IMS can be used with Microsoft Visual Basic bounded recordsets. This sample includes Visual Basic code and CICS COBOL code showing how to use bounded recordsets by calling into a CICS transaction program via LU 6.2 (Remote Environment CICS using LU 6.2). The Visual Basic code is in the `COMTI\BoundedRecorsdsets\VB` folder and demonstrates how to create a recordset and populate it with data to send to the mainframe. Note that there is no code that actually displays the data that comes back from the mainframe. A developer can put a breakpoint in the VB code using the debugger and use the immediate window to look at the data or insert further code to examine the data that is returned.

In the `COMTI\BoundedRecorsdsets\COBOL-CICS` folder, there is a COMTI type library (TLB file) that can be used with this sample. The type library is set up for accessing a transaction named GETI on the host. There is also sample COBOL code that can be compiled and linked on the mainframe side. The compiled code should be set up to run on the host as a transaction named GETI or the COMTI type library must be changed to reflect the name of the transaction if it is different.

Programming Specifics Sample

The Microsoft® COM Transaction Integrator (COMTI) for CICS and IMS supports a number of powerful features that are illustrated by these samples. In the `COMTI\ProgrammingSpecifics` folder, there is a complete Microsoft Visual Basic project that demonstrates the following features of COMTI:

- Returning a Recordset
- Variable Length Tables
- Handling REDEFINES Clauses
- Variably Sized Strings
- Handling FILLER
- Unbounded Recordsets
- In/Out Variable Length Table

The Visual Basic project contains comments with the Visual Basic source code that indicates which COBOL (*.cbl) file contains the associated COBOL code for the mainframe side. The sample type library is included for CICS using LU 6.2. There are seven methods defined in the type library. Check on the properties for each method and look at the **Host Names** tab to see what the Mainframe TP name is. The value of the mainframe TP name property can be changed to the name used when compiling the sample COBOL programs.

Programming Specifics CICSNonlink Sample

In the `COMTI\ProgrammingSpecifics\CICSNonlink` folder there is sample COBOL code showing how to receive more than 32K bytes of data in a single method call. This sample includes only the mainframe code (COBOL), and does not include the corresponding Visual Basic or Visual C++ code for the PC side. It is intended to demonstrate the receiving side logic of a COMTI Non-link server application. This COBOL program contains comments explaining what is being done in the code.

Programming Specifics TCP Sample

In the `COMTI\ProgrammingSpecifics\TCP` folder, there are sample COBOL Child Server programs that can be used for TCP/IP connections. The `Cicscs.cbl` code is a sample program for TCP using CICS with Concurrent Server (analogous to CICS using LU 6.2). The `Mscmtics.cbl` code is a sample program for CICS calling a Link-to program (using CICS DPL). The `Imsexpl.cbl` code is a sample program for using IMS in the Explicit mode. The `Imsimpl.cbl` code is a sample program for using IMS in the Implicit mode. There are similar sample programs included with the CedarBank tutorial, which directly reflect the CedarBank data being

passed.

Sync Level 2 Sample

The Microsoft COM Transaction Integrator (COMTI) for CICS and IMS supports the use of Sync Level 2 transactions. This sample includes COBOL source code illustrating transactional support (Sync Level 2) on the mainframe with CICS using LU 6.2. This sample only includes COBOL source code, which contains comments describing each of the code sections. The sample code demonstrates executing a Commit and identifying that a Rollback has been requested from COMTI. Please note that there is also related documentation in Knowledge Base article Q220967, Explanation of COMTI Metadata Elements. This article explains COMTI Metadata elements so that developers can better understand how to use Metadata to allow the COBOL program to initiate a Rollback of a transaction.