Microsoft Dynamics® GP

Continuum API Guide

Publication date

April 2009

Contents

Introduction

Continuum is the COM application programming interface (API) that is available for Microsoft Dynamics™ GP or any other Dexterity®-based applications. Tools that support COM automation, such as Visual Basic®, can use the Continuum API to interact with Microsoft Dynamics GP. Before using the Continuum API, review the information in this manual. It will help you decide whether Continuum API is suitable for your integration.

Prerequisites

The information in this manual will help you use the Continuum API to integrate with Microsoft Dynamics GP. It is assumed that you understand how to use the COM integration capabilities of the development tool you have chosen, and are familiar with the Microsoft Dynamics GP application.

Any development tool capable of interacting with COM applications can use the Continuum API. However, this documentation focuses on using Visual Basic .NET to create integrations with Continuum. If you are using a different development tool, the key concepts described here still apply. You will need to use syntax and development methods appropriate for the development tool you have chosen.

What's in this manual

The Continuum API Guide is designed to be a basic reference for the Continuum API. It also describes programming and deployment issues you will need to address for your Continuum integration. The manual is divided into the following parts:

- Part 1, **Getting Started**, explains the basics of a Continuum integration, and describes how to set up a Continuum project.
- Part 2, **Developing Integrations**, contains the information that you will need to create integrations using Continuum. For example, it introduces the sanScript language and explains how to use pass-through sanScript in your Continuum integrations. It also describes how to package your integrating applications for delivery to customers.
- Part 3, **Integration Examples**, describes sample applications that use Continuum to integrate with Microsoft Dynamics GP.
- Part 4, **The Continuum Integration Library**, describes the objects that are available through the Continuum API.

Symbols and conventions

To help you use this documentation more effectively, we've used the following symbols and conventions within the text to make specific types of information stand out.

Symbol	Description
💡	The light bulb symbol indicates helpful tips, shortcuts and suggestions.
⚠	Warnings indicate situations you should be especially aware of.
Margin notes summarize important information.	Margin notes direct you to other areas of the documentation where a given topic is explained.

Convention	Description
Part 1, **Getting Started**	Bold type indicates the name of a part.
Chapter 7, "Packaging"	Quotation marks indicate the name of a chapter.
Applying formats	Italicized type indicates the name of a section.
`set 'l_Item' to 1;`	This font is used for script examples.
Application Programming Interface (API)	Acronyms are spelled out the first time they're used.
TAB or ALT+M	Small capital letters indicate a key or a key sequence.

Part 1: Getting Started

This part describes the basics of the Continuum API, and explains how to set up a new Continuum project. The information is divided into the following areas:

- Chapter 1, "Integration Basics," provides an overview of how a Continuum integration works.
- Chapter 2, "Setting Up a Continuum Project," explains how to create a new project for a Continuum integration.

Be sure to review the information in this part of the documentation before you begin using Continuum. Understanding the basics of an integration will make learning the API easier.

Chapter 1: Integration Basics

Before creating applications that use the Continuum API to integrate with Microsoft Dynamics GP, you should have a basic understanding of how these integrations work and what the Continuum API provides. This information is divided into the following sections:

- *COM Automation*
- *The Continuum Integration Library*
- *Methods*
- *Triggers and callbacks*
- *Types of integrations*

COM Automation

An object is a combination of code and data that contains information about an application or an item in the application, such as a field or window. COM Automation is an industry-standard technology that allows applications to provide access to objects in the application. An application that provides access to its objects is called an automation server. An application that accesses objects from an automation server is called an automation client.

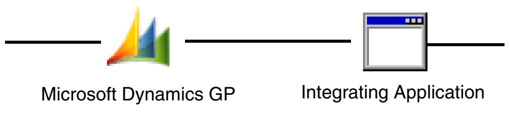
The Microsoft Dynamics GP runtime engine has an object that describes the application and another object that describes a field in the application. Code has been added to the runtime engine that allows it to act as a server. This means that Microsoft Dynamics GP makes its objects available to other applications, allowing those applications to interact with Microsoft Dynamics GP through the COM programming interface.

The Continuum Integration Library

*The objects in the Continuum Integration Library are described in detail in Part 4, **The Continuum Integration Library**.*

The Continuum Integration Library is the code in the Microsoft Dynamics GP application that describes the objects that can be accessed through the COM programming interface. Your integrating application will use the properties and methods for these objects when it interacts with Microsoft Dynamics GP.

Microsoft Dynamics GP acts as an automation server. It makes its objects available to other applications through the COM programming



The integrating application acts as an automation client, accessing objects in Microsoft Dynamics GP.

The Microsoft Dynamics GP application objects provide two basic actions that enable integration. First, they allow actions to be executed in Microsoft Dynamics GP, such as pushing a button or setting the value of a window field. Second, they notify the integrating application when certain events occur in Microsoft Dynamics GP, such as when a form opens or the value in a field changes.

Methods

To make things happen in the Microsoft Dynamics GP application through the Continuum API, your integrating application will use the methods provided. Most of the methods you will use are associated with the Application object, such as the **Activate method**, which is used the bring the Microsoft Dynamics GP application to the foreground.

The method you will use most is the **ExecuteSanScript method**, which allows you to execute sanScript code in the Microsoft Dynamics GP application. SanScript is the language used to build the Microsoft Dynamics GP application. It provides numerous commands to perform actions like hiding and showing fields, opening and closing windows, and so on. You will learn more about using sanScript code in Chapter 3, "Pass-through sanScript."

Triggers and callbacks

An application that integrates through the Continuum API can watch for and respond to specific events within the Microsoft Dynamics GP application. These events, called *triggers*, are defined by your integrating application. You will use methods from the Continuum API, such as the **RegisterFocusTrigger method**, to specify which actions to watch for within Microsoft Dynamics GP.

When a trigger is activated, a corresponding callback method that you specified is run in your integrating application. Typically, this callback method will execute code in response to the action that occurred in Microsoft Dynamics GP.

For example, you could use the **RegisterFocusTrigger method** to register a trigger that is activated with the Toolbar window in Microsoft Dynamics GP is closed. When the Microsoft Dynamics GP application is shut down, the Toolbar window is closed and the trigger is activated. The callback method you specified will be run at that time. Typically, the callback method would close the integrating application, since Microsoft Dynamics GP is no longer running.

Types of integrations

Three types of integrations can be created with Continuum:

Interface-level integrations Those in which the integrating application interacts with or manipulates the user interface in Microsoft Dynamics GP. These are the easiest integrations to create.

Database-level integrations Those in which the integrating application reads from and writes to the Microsoft Dynamics GP database. To do these, you must understand the Microsoft Dynamics GP database. The Continuum API provides limited support for these types of integrations, so keep in mind that other integration methods might be more appropriate.

Process-level integrations Those in which the integrating application updates information whenever a Microsoft Dynamics GP process, such as posting, is executed. These integrations require a thorough understand the process you're integrating with in the Microsoft Dynamics GP application.

Chapter 2: Setting Up a Continuum Project

To function properly, your integrating application must contain the code to set up the automation link to the Microsoft Dynamics GP application. This portion of the documentation describes what is required in your code, and provides an example that shows how to set up a Continuum project in Visual Basic .NET.

Information is divided into the following sections:

- *Required items*
- *Visual Basic project*
- *TemplateMain module*
- *Callback class*

Required items

To allow your integrating application to access the COM API in Microsoft Dynamics GP, you must have the following:

Properly configured project

The development tool you are using may require special project or application settings to allow access to COM resources. For example, you make need to add a reference to the type library (named Dex.tlb) that defines the Continuum interface. Check the documentation for information about the development tool you are using.

Initialization code

Your integration must contain initialization code to access the Continuum API. This initialization code does the following:

- Defines a variable for an instance of the Microsoft Dynamics GP application object.
- Creates a reference to the Microsoft Dynamics GP application.
- Specifies how to handle cross-thread calls, for Visual Studio 2005 and later. Typically, the check for illegal cross-thread calls is suppressed.
- Defines a variable for an instance of the callback class (if you will be using triggers and callbacks).

This initialization code must be run before you can use any of the methods or properties defined in the Continuum API..

Callback class

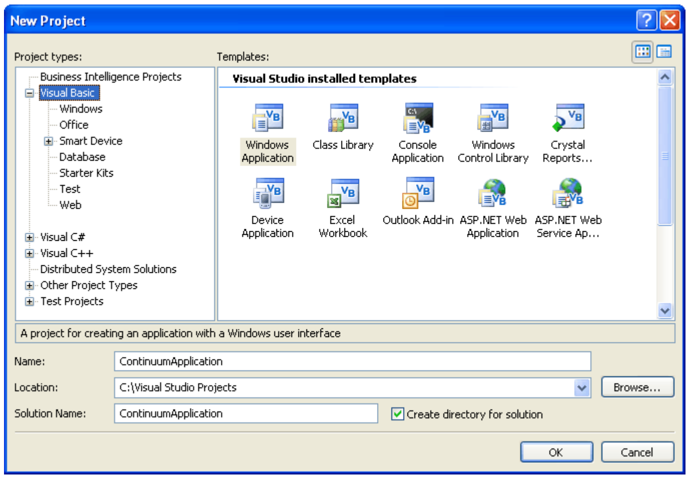
If you will be using triggers and callbacks with your integration, you must define a callback class that will contain the methods that are run in response to events occurring in the Microsoft Dynamics GP application.

If you create a callback class, remember that the initialization code must define a variable that stores an instance of the callback class. You may also have to adjust the accessibility level for the class (such as making it public) so the callback methods in the class can be executed by the Continuum API.

Visual Basic project

The following procedure describes how to create a Visual Basic .NET project that will access the Continuum API. If you are using another development tool, adjust the steps accordingly.

1. **Create a new application project.**
   From the File menu, choose to create a new project. Select Visual Basic as the project type, and Windows Application as the template to use.



New Project

Project types:
- Business Intelligence Projects
- Visual Basic
  - Windows
  - Office
  - Smart Device
  - Database
  - Starter Kits
  - Test
  - Web
- Visual C#
- Visual C++
- Distributed System Solutions
- Other Project Types
- Test Projects

Templates:

Visual Studio installed templates

A project for creating an application with a Windows user interface

Name: ContinuumApplication

Location: C:\Visual Studio Projects

Solution Name: ContinuumApplication

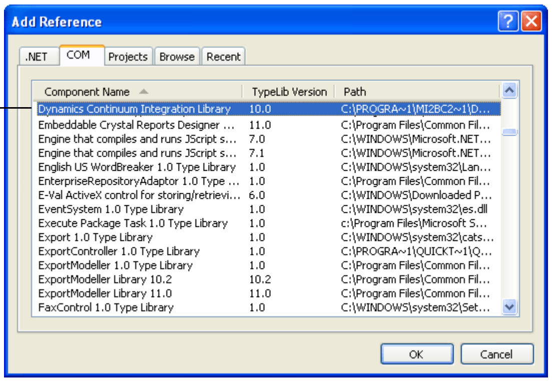
✓ Create directory for solution

2. **Add a reference to the Continuum type library.**
   The Continuum type library describes the methods and properties available in the Continuum API for Microsoft Dynamics GP. The type library file for Continuum is named **Dex.tlb**, and is located in the same folder as the Microsoft Dynamics GP installation.

   When Microsoft Dynamics GP is installed, the Continuum type library is registered on the system. This allows development tools like Visual Basic to easily find it, so you won't have to manually locate the file.

   From the Project menu, choose Add Reference. Select COM as the type of reference to add. Locate **Dynamics Continuum Integration Library** in the list and click Select. If multiple versions are listed, select the one for the version of Microsoft Dynamics GP you are integrating with.

Choose Dynamics Continuum Integration Library.



Add Reference

Component Name	TypeLib Version	Path
Dynamics Continuum Integration Library	10.0	C:\PROGRA~1\MI2BC2~1\D...
Embeddable Crystal Reports Designer ...	11.0	C:\Program Files\Common Fil...
Engine that compiles and runs JScript s...	7.0	C:\WINDOWS\Microsoft.NET...
Engine that compiles and runs JScript s...	7.1	C:\WINDOWS\Microsoft.NET...
English US WordBreaker 1.0 Type Library	1.0	C:\WINDOWS\system32\Lan...
EnterpriseRepositoryAdaptor 1.0 Type ...	1.0	C:\Program Files\Common Fil...
E-Val ActiveX control for storing/retrievi...	6.0	C:\WINDOWS\Downloaded P...
EventSystem 1.0 Type Library	1.0	C:\WINDOWS\system32\es.dll
Execute Package Task 1.0 Type Library	1.0	c:\Program Files\Microsoft S...
Export 1.0 Type Library	1.0	C:\WINDOWS\system32\cats...
ExportController 1.0 Type Library	1.0	C:\PROGRA~1\QUICKT~1\Q...
ExportModeller 1.0 Type Library	1.0	C:\Program Files\Common Fil...
ExportModeller Library 10.2	10.2	C:\Program Files\Common Fil...
ExportModeller Library 11.0	11.0	C:\Program Files\Common Fil...
FaxControl 1.0 Type Library	1.0	C:\WINDOWS\system32\Set...

Click OK to add the reference to the project.

By adding this reference, you're telling the development environment about the COM interface you intend to use. This allows features like IntelliSense to help with writing code, and allows you to use the Object Browser within Microsoft Visual Studio® to view information about the Continuum API.

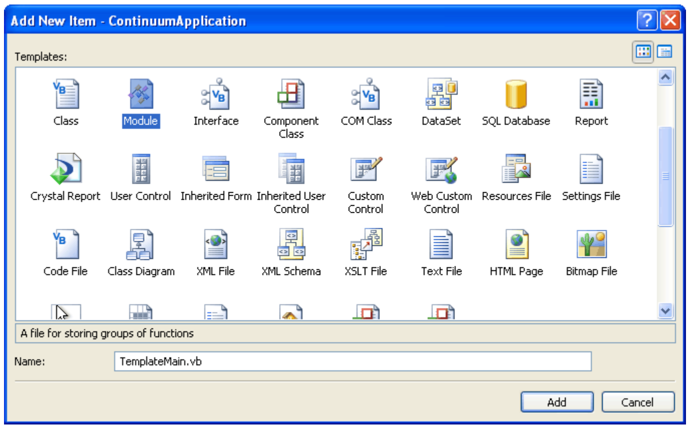
Referencing the type library may be required for some development tools to properly access the Continuum API.

If you add the reference to the type library, and use fully-qualified references to the objects available in the Continuum API, you will need to include the "interop" file that is generated to allow Visual Basic .NET to access the COM interface for Microsoft Dynamics GP.

TemplateMain module

The following procedure describes how to add a TemplateMain module to your Visual Basic .NET project. This code module contains the initialization code for the Continuum API.

1. **Add a module to the project.**
From the Project menu, choose Add Module. The Add New Item dialog will be displayed. Choose to add a module.



A file for storing groups of functions

Name: TemplateMain.vb

Name the module TemplateMain.vb, or some other name indicating the module contains the initialization code, and then click Add.

2. **Add initialization code to the code module.**
Add the following initialization code to the code module you created.

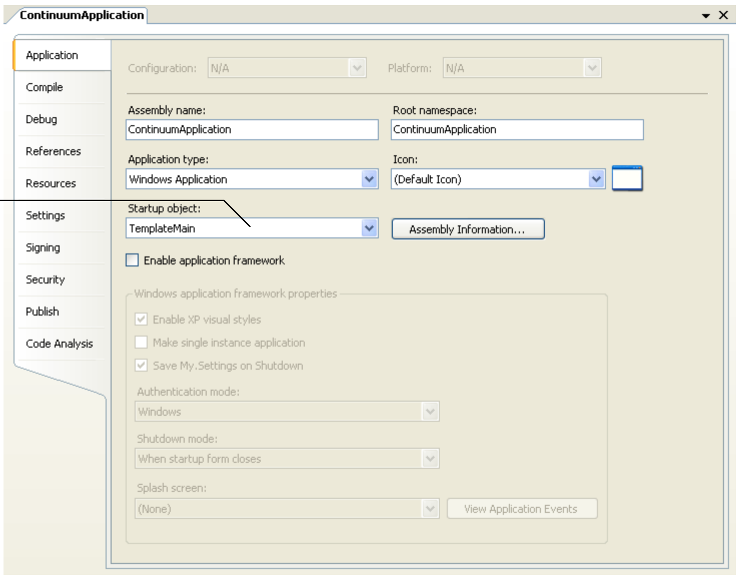
```
Public GPApp As Dynamics.Application
Public GPCallback As New GPCallbackClass()

Public Sub Main()
    'Create the GP application object
    GPApp = CreateObject("Dynamics.Application")
    On Error GoTo 0

    If GPApp Is Nothing Then
        MsgBox("Failed to create the application object")
    End
    End If

    ' Prevent thread warnings for callbacks
    System.Windows.Forms.Control.CheckForIllegalCrossThreadCalls = False
End Sub
```

This initialization code defines a variable for the Microsoft Dynamics GP application object. Notice that the **CreateObject()** method is used to create an instance of the Microsoft Dynamics GP application.

This initialization code also creates an instance of the GPCallbackClass, which is required if you will be using triggers and callbacks with your Continuum integration. If you won't be using callbacks, you can omit this line of the initialization code.

3. **Set the Startup object for the project.**
Within your Visual Basic project you must specify which code will run when your integration starts. Since the Continuum initialization code is contained in the TemplateMain module, specify this module as the Startup object.

Set the Startup object to TemplateMain.

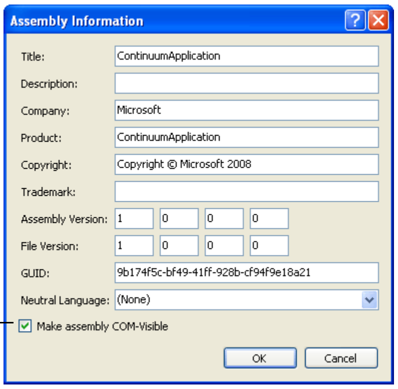


Configuration: N/A
Platform: N/A
Assembly name: ContinuumApplication
Root namespace: ContinuumApplication
Application type: Windows Application
Icon: (Default Icon)
Startup object: TemplateMain
☐ Enable application framework
Windows application framework properties
✓ Enable XP visual styles
☐ Make single instance application
✓ Save My.Settings on Shutdown
Authentication mode: Windows
Shutdown mode: When startup form closes
Splash screen: (None)

You must unmark the Enable application framework option to specify TemplateMain as the startup object.

If you need to specify a different startup object, such as a form or other code module, you must move the initialization code for Continuum to that location.

4. **Make the assembly COM-visible (if required).**
You may need to make the assembly for your integration COM-visible for triggers to register and run properly. To do this, click Assembly Information in the Application Properties. Mark Make assembly COM-Visible and click OK.



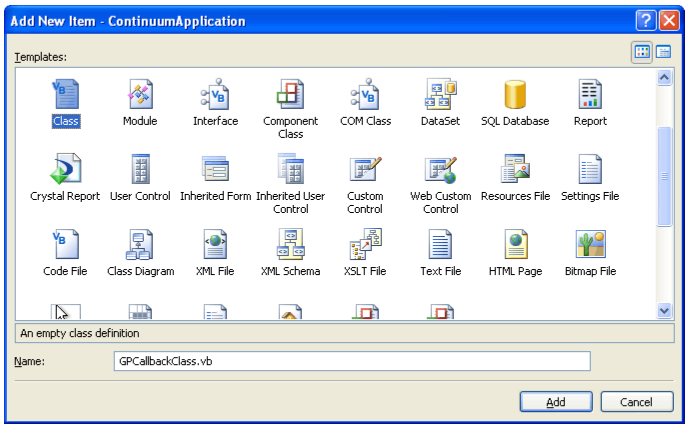
Title: ContinuumApplication
Description:
Company: Microsoft
Product: ContinuumApplication
Copyright: Copyright © Microsoft 2008
Trademark:
Assembly Version: 1 0 0 0
File Version: 1 0 0 0
GUID: 9b174f5c-bf49-41ff-928b-cf94f9e18a21
Neutral Language: (None)
✓ Make assembly COM-Visible

Mark this option to make the assembly visible to COM.

Callback class

If you will be using triggers and callbacks for your Continuum integration, you need to create a callback class that will contain the callback methods. To do this, complete the following procedure.

1. **Add a class to the project.**
   From the Project menu, choose Add Class. The Add New Item dialog will be displayed. Choose to add a class.



An empty class definition

Name: GPCallbackClass.vb

   Name the class GPCallbackClass.vb, or some other name indicating the class contains the callback methods for your integration, and then click Add.

2. **Update the initialization code to create the class instance.**
   In the initialization code for your integration, verify that you have created an instance of the class that contains the callback methods. For instance, if the callback class is named GPCallbackClass, the following line is required in the initialization code:

```
Public GPCallback As New GPCallbackClass()
```

If you used a different name for the callback class, be sure to use the new name in the initialization code.

Part 2: Developing Integrations

This portion of the documentation contains information about developing integrations with Continuum. The information is divided into the following areas:

- Chapter 3, "Pass-through sanScript," describes how to use sanScript code in your Continuum integration.
- Chapter 4, "Passing Parameters," explains how to pass information between sanScript and your Continuum integration code.
- Chapter 5, "Database Integrations," describes how your Continuum integration can interact directly with the Microsoft Dynamics GP database.
- Chapter 6, "Programming Techniques," describes additional issues you need to be aware of when you create integrations using the Continuum API.
- Chapter 7, "Packaging," explains how to package your integrating application once it's complete.

Chapter 3: Pass-through sanScript

The Application object in the Continuum API contains a method that allows you to pass sanScript code into the Microsoft Dynamics GP runtime engine, which will compile and execute it.

We recommend that you be familiar with the Dexterity development system and the sanScript language if you want to use sanScript from within your Continuum integration. If you are not familiar with Dexterity, you can use information in the sanScript supplement (SanScriptSupplement.pdf) included with Continuum to learn about core sanScript functionality.

The following items are discussed:

- *Writing and executing scripts*
- *Looking up names*
- *Debugging scripts*

Writing and executing scripts

When sanScript code is passed into Microsoft Dynamics GP to be compiled and executed, it is actually run as if it were a procedure script in the Microsoft Dynamics GP application. This means that the pass-through script has characteristics similar to those of procedures. These characteristics include:

- The pass-through script runs in the foreground. When it is running, no other processing occurs in Microsoft Dynamics GP.

- All resources referenced by the pass-through script must have their names fully qualified to be referenced properly.

- The pass-through script has access to its own table buffer for each table in the application.

- Unlike procedures, parameters can't be passed into the pass-through script. Instead, parameter values must be set and retrieved using OLE methods. This is described in Chapter 4, "Passing Parameters."

Writing scripts

In most cases, the pass-through sanScript code you write will look the same as the sanScript code used in Dexterity-based applications. One exception is when you include a literal string in your pass-through sanScript code. In ordinary sanScript, a literal string is a string value enclosed in quotation marks. With pass-through sanScript, you must enclose a literal string in two sets of quotation marks so the Visual Basic compiler will properly interpret your pass-through sanScript code. For example, the following sanScript code sets the value of the first_name variable.

```
set first_name to "Steve";
```

To use this code in pass-through sanScript, you must enclose the literal string "Steve" in two sets of quotation marks to be interpreted properly.

```
set first_name to ""Steve"";
```

Executing scripts

*For more information, refer to the description of the **ExecuteSanScript** method in Chapter 11, "Application Object."*

You will use the **ExecuteSanScript method** for the Microsoft Dynamics GP application object to compile and execute your sanScript code. This method takes the sanScript source code as a string and passes it to the Microsoft Dynamics GP runtime engine. The Microsoft Dynamics GP runtime engine will attempt to compile and execute the sanScript code. If the code can't be compiled, a compiler error will be returned to the **ExecuteSanScript method**. Any error generated when the sanScript code runs will be displayed by Microsoft Dynamics GP.

Your integrating application must contain the initialization code for Continuum in order for pass-through sanScript to work properly.

Script example

To show how pass-through sanScript works, the following sanScript code opens the Receivables Transaction Entry window and sets the Document Type field to Service/Repairs.

```
{This command opens the Receivables Transaction Entry window.}
open form RM_Sales_Entry;

{This command sets the Document Type drop-down list.}
set 'Document Type' of window RM_Sales_Entry of form RM_Sales_Entry to 4;
```

Once the code has been written, you must place it into your Visual Basic application so it can be passed to the Microsoft Dynamics GP runtime engine. The following Visual Basic procedure uses the **ExecuteSanScript method** to pass the sanScript code to Microsoft Dynamics GP.

```
Private Sub Receivables_Click()

    'Variables used for return values
    Dim ErrVal As Integer
    Dim error_msg As String

    ErrVal = GPApp.ExecuteSanScript("open form " & _
    "RM_Sales_Entry; set 'Document Type' of window " & _
    "RM_Sales_Entry of form RM_Sales_Entry to 4;", _
    error_msg)

    If ErrVal <> 0 Then
        'A compiler error occurred. Display the error.
        MsgBox error_msg
    End If

End Sub
```

Note the sanScript code in the first parameter for the **ExecuteSanScript method**. The sanScript code must be contained in a single string. For short scripts, you can do this by including the code on a single line. For longer scripts, you may want to break the script into smaller strings that are more manageable, as was done for this example.

Looking up names

Any pass-through sanScript code you write must use the appropriate names to access resources in Microsoft Dynamics GP. It can be difficult to find the correct names for resources in Microsoft Dynamics GP. To make this easier, the Continuum interface provides two "wizard mode" methods that are used to look up names in the Microsoft Dynamics GP application. The **StartWizardMode method** and **StopWizardMode method** allow you to click on a resource in Microsoft Dynamics GP and return its name for use in your pass-through sanScript.

Chapter 10, "Name Wizard," describes a sample application that shows how to use these two methods to retrieve names from Microsoft Dynamics GP. You may want to compile and use this sample as a development tool when writing pass-through sanScript for your Continuum integration.

Debugging scripts

Once you have written sanScript code and placed it into your Visual Basic application, you can send it to Microsoft Dynamics GP to be executed.

Compiler errors

When sanScript code is sent to Microsoft Dynamics GP, the runtime engine will attempt to compile the code. If the code can't be compiled, a compiler error will be returned to the **ExecuteSanScript method**. The second parameter of this method will contain the compiler error message. Use the message to debug your sanScript code.

Runtime errors

If the sanScript code is successfully compiled, it will be executed by Microsoft Dynamics GP. If your sanScript code attempts to perform an operation that isn't allowed, a runtime error will occur. A runtime error will display a message in Microsoft Dynamics GP that describes the error that occurred.

Chapter 4: Passing Parameters

In some cases, you may need to pass values into or return values from your pass-through sanScript. This portion of the documentation describes how to perform these actions. The following topics are discussed:

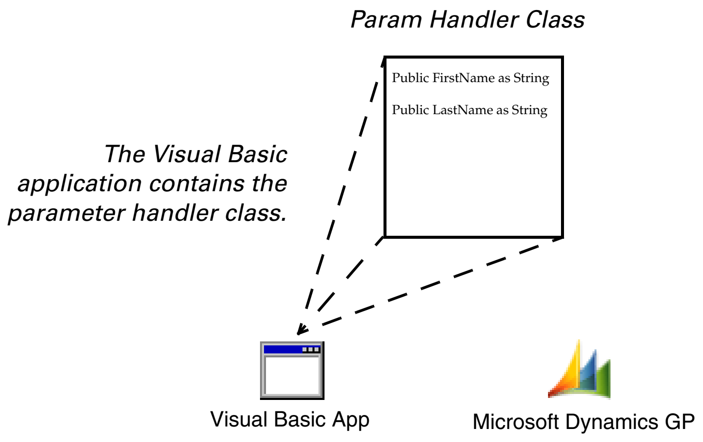
- *Parameter handler*
- *Setting and getting properties*
- *Running methods*

Parameter handler

To pass parameters between Visual Basic and your pass-through sanScript, you must use a parameter handler. To create a parameter handler, you first add a *parameter handler class* to your Visual Basic application. Then you create a *parameter handler object* based upon the new class.

Parameter handler class

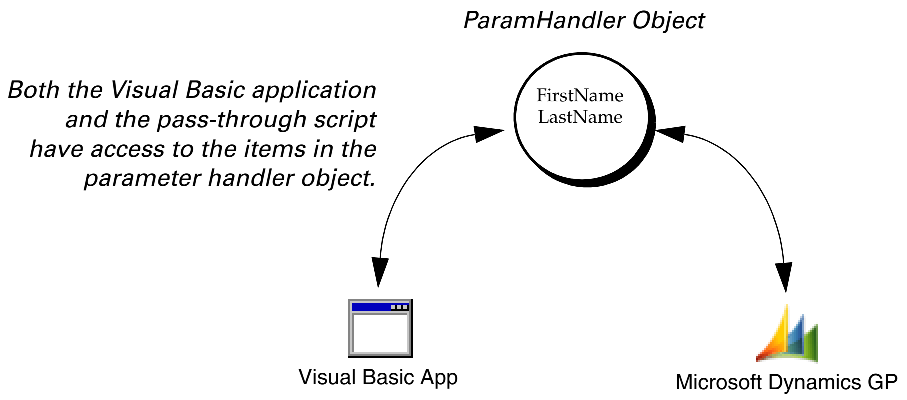
The parameter handler class contains properties and methods that define what type of values you want to pass between your Visual Basic application and pass-through sanScript. For example, if you wanted to pass a first and last name into your pass-through sanScript, the parameter handler class would contain two string properties – one for the first name and one for the last name.


Param Handler Class
Public FirstName as String
Public LastName as String
The Visual Basic application contains the parameter handler class.
Visual Basic App
Microsoft Dynamics GP

For the Continuum API, only string values can be passed as parameters.

Parameter handler object

When your Visual Basic application runs, it must create a parameter handler object based upon the parameter handler class. This object contains the properties and methods you defined in the parameter handler class. Both the Visual Basic application and the pass-through sanScript have access to the properties and methods in the parameter handler object.


ParamHandler Object
FirstName
LastName
Both the Visual Basic application and the pass-through script have access to the items in the parameter handler object.
Visual Basic App
Microsoft Dynamics GP

When you've created the parameter handler object, you must use the SetParamHandler method of the Microsoft Dynamics GP application object to specify which object in the Visual Basic application is being used as the parameter handler. This allows the pass-through sanScript to know which object to use when it sets or retrieves parameter values.

Continuing the previous example, you would create a parameter handler object based on the parameter handler class. This object would contain properties for the first name and last name. You would then use the **SetParamHandler method** to tell the pass-through sanScript what object you were using as the parameter handler. Then both the Visual Basic application and the pass-through sanScript would have access to the properties in the parameter handler object.

Example 1

The following example illustrates how to write and set up a parameter handler class, how to create a parameter handler object, and how to specify the object that will be used as the parameter handler.

Parameter handler class

In this example, the parameter handler class contains a CustomerNumber parameter and a CustomerName parameter. A new class module named ParamHandlerClass was added to the Visual Basic project. The following declarations were added to this class to create the CustomerNumber and CustomerName properties.

```
Public Class ParamHandlerClass
    Public CustomerNumber As String
    Public CustomerName As String
End Class
```

Parameter handler object

Once the parameter handler class is defined, the Visual Basic application must create a parameter handler object based upon the class. Typically, this is done in the TemplateMain module, where other global variables and constants are defined. The following code was added at the beginning of TemplateMain to create the parameter handler object.

```
'Create the parameter handler object
Public ParamHandler As New ParamHandlerClass()
```

Finally, the **SetParamHandler method** for the Application object is used to specify which object in Visual Basic will be used as the parameter handler. This is necessary so that the pass-through sanScript code knows which object to use when it sets and gets properties. The following code was added to the initialization code for the Visual Basic project. It specifies which object to use as the parameter handler.

```
'Set the parameter handler object
Dim ErrVal As Integer
ErrVal = GPApp.SetParamHandler(ParamHandler)
```

Setting and getting properties

Once you have created the parameter handler object, you can use it to pass values between Visual Basic and pass-through sanScript.

Visual Basic

To set and get properties from Visual Basic, you interact with the parameter handler object the same way you would with any other object. For example, the following Visual Basic code would set the FirstName property in the ParamHandler class:

```
ParamHandler.FirstName = "Steve"
```

The following line of Visual Basic code retrieves the value of the LastName property in the ParamHandler class:

```
Dim LName As String
LName = ParamHandler.LastName
```

Pass-through sanScript

To set and get properties from pass-through sanScript, you must use two functions from sanScript's OLE function library. To set properties, use the **OLE_SetProperty()** function. The syntax and parameters of this function are as follows:

OLE_SetProperty(*property_name*, *value_string***)**

- *property_name* – The name of the property in the parameter handler object whose value you want to set.

- *value_string* – The string value to which you want to set the property.

The return value of this function is a boolean that indicates whether the function succeeded; true indicates the function succeeded, false indicates it didn't.

As an example, the following sanScript code sets the FirstName property in the ParamHandler class:

```
local boolean err_val;

set err_val to OLE_SetProperty("FirstName", "Steve");
```

To get properties, use the **OLE_GetProperty()** function. The syntax and parameters of this function are as follows:

OLE_GetProperty(*property_name*, *value_string***)**

- *property_name* – The name of the property in the parameter handler object whose value you want to get.
- *value_string* – The string variable that will contain the property's value.

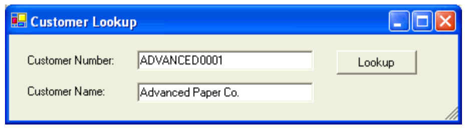
The return value of this function is a boolean that indicates whether the function succeeded; true indicates the function succeeded, false indicates it didn't.

As an example, the following sanScript code gets the value of the LastName property in the ParamHandler class:

```
local boolean err_val;
local string last_name;

set err_val to OLE_GetProperty("LastName", last_name);
```

Example 2

The following example illustrates how to pass values between Visual Basic and pass-through sanScript. This example is based on Example 1, described earlier in this chapter. The Customer Lookup window, shown in the following illustration, is used to look up the name of a customer based upon the Customer Number.



The following Visual Basic code is attached to the Lookup button. This code passes the Customer Number into the pass-through sanScript, which looks up and returns the corresponding Customer Name to the Visual Basic application.

```
Private Sub Lookup_Click()

    Dim err_val As Integer
    Dim error_msg As String

    'Set the CustomerNumber parameter
    ParamHandler.CustomerNumber = CustomerNumber.Text

    'Use pass-through sanScript to retrieve the Customer Name
    err_val = DynamicsApp.ExecuteSanScript( _
    "local boolean err_val; local string cust_num, cust_name; " & _
    "err_val = OLE_GetProperty(""CustomerNumber"",cust_num); " & _
    "'Customer Number' of table RM_Customer_MSTR = cust_num; " & _
    "get table RM_Customer_MSTR; " & _
    "if err() = OKAY then " & _
    " err_val = OLE_SetProperty(""CustomerName"",'Customer Name' " & _
    " of table RM_Customer_MSTR); " & _
    "else " & _
    " err_val = OLE_SetProperty(""CustomerName"",""Not Found""); " & _
    "end if;", error_msg)

    'Retrieve and display the CustomerName parameter
    CustomerName.Text = ParamHandler.CustomerName

End Sub
```

Running methods

From within pass-through sanScript, you can run methods that have been defined in the parameter handler object. To do this, you use the **OLE_RunMethod()** function. The syntax and parameters of this function are as follows:

OLE_RunMethod(*method_name, value_string***)**

- *method_name* – The name of the method you wish to run in the parameter handler object.
- *value_string* – The string value you want to pass to the method.

The return value of this function is a boolean that indicates whether the function succeeded; true indicates the function succeeded, false indicates it didn't.

Executing a method in the parameter handler object is useful if you have private data members whose values are set through the use of property procedures.

As an example, the following sanScript code calls the SetName method in the parameter handler object:

```
local boolean err_val;

set err_val to OLE_RunMethod("SetName", "Steve").
```

Chapter 5: Database Integrations

Your Visual Basic application can be informed of various database events that occur in Microsoft Dynamics GP. You must register triggers to indicate which database events your application will be notified of. When a database event occurs for which a trigger has been registered, a procedure in your application will run, allowing it to respond to the event. Information is divided into the following topics:

- *Registering database triggers*
- *Database trigger reference*
- *Accessing table data*

Registering database triggers

Refer to Chapter 11, "Application Object," for a complete description of the ***RegisterDatabase Trigger method****.*

To register a database trigger, you will use the **RegisterDatabaseTrigger method** from the Continuum Integration Library. When you register a database trigger, you specify which table in Microsoft Dynamics GP to monitor, which database operation or operations you want to be notified of, and which method you want to run in the callback class in your application.

For example, the following Visual Basic code registers a database trigger for the RM_Customer_MSTR table. The notification will occur each time a record is added to the table. The RMCustAdd procedure in the GPCallback class will run when the notification occurs.

```
Dim ErrVal As Integer
ErrVal = GPApp.RegisterDatabaseTrigger( _
"table RM_Customer_MSTR", "", 4, GPCallback, "RMCustAdd")
If ErrVal <> 0 Then
    MsgBox "Unable to register the database notification."
End If
```

Typically, you will place the code to register database triggers in the same location as the code that initializes Continuum.

In the previous example, the database trigger will run *anytime* a record is added to the RM_Customer_MSTR table. In some cases, you may want the trigger to occur when only a specific form performs the database operation. You can use the second parameter of the **RegisterDatabaseTrigger method** to specify which form must perform the database operation that causes the trigger to occur.

For example, the following Visual Basic code registers a database trigger for the RM_Customer_MSTR table. The trigger will occur each time a record is read from the table by the RM_Customer_Maintenance form. The RMCustRead procedure in the GPCallback class will run when the trigger occurs. The trigger will *not* occur when other forms or procedures read records from the RM_Customer_MSTR table.

```
Dim ErrVal As Integer
ErrVal = GPApp.RegisterDatabaseTrigger( _
"table RM_Customer_MSTR", "form RM_Customer_Maintenance", 3, _
GPCallback, "RMCustRead")
If ErrVal <> 0 Then
    MsgBox "Unable to register the database notification."
End If
```

Database trigger reference

The following table lists the database operations for which you can register triggers. It also lists the integer value that corresponds to the database operation.

Operation	Value	Description
Read without lock	1	Occurs when Microsoft Dynamics GP reads a record in the table without locking it.
Read with lock	2	Occurs when Microsoft Dynamics GP reads a record in the table with either a passive or active lock.
Add	4	Occurs when Microsoft Dynamics GP adds a new record to the table.
Update	8	Occurs when Microsoft Dynamics GP updates a record in the table.
Delete	16	Occurs when Microsoft Dynamics GP deletes a record from the table.

You can add these values together to run a trigger for more than one type of database operation. For instance, the integer "3" registers a single trigger that will run for all types of database read operations.

Database triggers occur for only successful *database operations. If a database operation fails in Microsoft Dynamics GP, the database trigger will not occur.*

In a typical database-level integration that keeps integrating application data synchronized with Microsoft Dynamics GP, you will register the following database triggers:

- **Read** operations – In most cases, a single trigger can deal with both types of read operations. When the read trigger occurs, the integrating application reads the appropriate data corresponding to the record read by Microsoft Dynamics GP. You will often restrict a database read trigger to a specific form in Microsoft Dynamics GP.

- **Add** and **Update** operations – In most cases, a single trigger can handle when a new record has been added to a Microsoft Dynamics GP table or when an existing record has been updated. When the trigger occurs, the integrating application adds or updates the data corresponding to the data written to the Microsoft Dynamics GP table.

- **Delete** operations – This trigger handles when Microsoft Dynamics GP deletes a record from a table. When the trigger occurs, the integrating application deletes the data corresponding to the record deleted by Microsoft Dynamics GP.

In all of these database triggers, the method that runs in response to the database trigger must be able to ascertain what data was being manipulated by Microsoft Dynamics GP. The next section describes how to work with Microsoft Dynamics GP data when database triggers occur.

Accessing table data

When a database trigger occurs, your integrating application must be able to ascertain what data in Microsoft Dynamics GP was read, added, updated or deleted. You will use the **GetDataValue method** from the Continuum Integration library to find out what record was manipulated.

Using GetDataValue

*Refer to Chapter 11, "Application Object," for a complete description of the **GetDataValue method**.*

In the callback method for the database trigger, you need to ascertain what record in the table was being accessed when the database trigger occurred. To do this, you will use the **GetDataValue method** from the Continuum Integration library.

When you use the **GetDataValue method** in the callback, you don't need to fully qualify the location of the table buffer you're accessing. By default, the **GetDataValue method** will access the appropriate table buffer for the table that caused the database trigger to occur.

For example, the following Visual Basic code is included in the callback method that runs in response to a database read operation on the RM_Customer_MSTR table. The **GetDataValue method** retrieves the value of the Customer Number field so the Visual Basic application knows which record was read.

```
Dim CustNumber As String
CustNumber = GPApp.GetDataValue( _
"'Customer Number' of table RM_Customer_MSTR")
```

Notice that the location of the RM_Customer_MSTR table buffer is not specified. The **GetDataValue method** automatically knows the specific table buffer to access, based upon the Microsoft Dynamics GP form or procedure that performed the database operation and caused the trigger to be run.

Finding field names

When using the **GetDataValue method**, you need to specify the name of the table field from which you want to retrieve data. The easiest way to find field names is to use the Table Descriptions window in the Microsoft Dynamics GP Resource Descriptions tool. The Table Descriptions window lists the fields that are part of each table. Use the names in the Field column when specifying a field for the **GetDataValue method**.

Chapter 6: Programming Techniques

This portion of the documentation describes several programming techniques and issues that you should be aware of when you create integrating applications with the Continuum API. The following topics are discussed:

- *Specifying the current product*
- *Starting integrating applications*
- *Retrieving data from the Microsoft Dynamics GP application*
- *Working with scrolling windows*
- *Managing cross-thread calls*
- *User Account Control (UAC)*

Specifying the current product

The architecture of the Microsoft Dynamics GP application allows the main product and multiple integrating products to operate together at the same time. When you issue commands with Continuum, you must specify which product you will be interacting with for the commands to function properly. You can do this with the **CurrentProduct property** or the **CurrentProductID property**.

By default the Continuum API will issue commands to the main product (Microsoft Dynamics GP). If you change the current product, all subsequent commands issued through the Continuum API will be run in the context of the product you specified. It's a good practice to always specify the current product before issuing commands with the Continuum API.

You can specify the current product by name with the **CurrentProduct property**, or by dictionary ID with the **CurrentProductID property**. The values for these can be found in the launch file for Microsoft Dynamics GP. They are also returned by the Name Wizard sample application described in Chapter 10.

Because the names for products can change, we recommend that you use the Product ID when specifying the current product for the Continuum API.

Starting integrating applications

In most cases, you will want your integrating application to start automatically when Microsoft Dynamics GP starts. The Microsoft Dynamics GP application *must* be running before an integrating application will be able to issue any commands or register any triggers.

The LAUNCHER.CNK file included with Continuum is a Dexterity-based application that integrates with Microsoft Dynamics GP. It is used to start other applications that integrate with the Microsoft Dynamics GP application through Automation. To install the Application Launcher application, copy the LAUNCHER.CNK file to the same location as Microsoft Dynamics GP. When you start the Microsoft Dynamics GP application, you will receive a message asking whether you want to include new code. Click Yes to install the application. Then exit the Microsoft Dynamics GP application.

To have the Application Launcher start your integrating application each time you start Microsoft Dynamics GP, you must add an entry to the DEX.INI file located in the same folder as the Microsoft Dynamics GP application. The entry must have the following form:

```
OLE_Applicationnumber=pathname
```

Substitute an integer for *number*, beginning with 1. Additional applications you want to start should use 2, 3, and so on. There can't be any gaps in the sequence. For *pathname*, substitute the complete path to your application. This path should be in the native Windows format.

As an example, the following entry could be added to the DEX.INI file to start the Additional Information Window sample application:

```
OLE_Application1=C:\GP\ADDLINFO.EXE
```

If you also wanted to start the Field Defaulter sample application, the following entry could be added:

```
OLE_Application2=C:\GP\FLDDFLTR.EXE
```

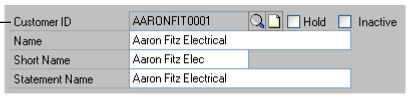
Retrieving data from the Microsoft Dynamics GP application

A common integration scenario is to register a notification so your application will be notified when a Microsoft Dynamics GP field changes. The callback procedure will then retrieve that field's value and the values of other fields in the window. As described, this scenario would appear to work properly. But, based upon how notifications actually work, you may not get the results you expect.

A notification occurs as soon as the control's value changes. This causes the callback procedure in the integrating application to be run. Other controls in the Microsoft Dynamics GP window that you think should have been updated because the first control changed won't have changed yet. If the callback procedure retrieves values for those controls, it will retrieve their previous values, not the updated values. The following example illustrates this scenario.

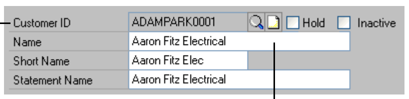
In the Microsoft Dynamics GP application Customer Maintenance window, there is a Customer ID. When the Customer ID changes, other values in the window are updated. Assume an integrating application registers to be notified when the Customer ID value changes. In the callback for the notification, the integrating program retrieves the Customer ID and the Name. You expect the Customer ID and its corresponding Name will be retrieved, but this isn't what occurs. When the Customer ID changes, the notification occurs before the Name value can be updated. Thus, the new Customer ID and the *old* Name are retrieved by the callback procedure.

A notification is registered for the Customer ID.



Customer ID	AARONFIT0001
Name	Aaron Fitz Electrical
Short Name	Aaron Fitz Elec
Statement Name	Aaron Fitz Electrical

When the Customer ID value changes, the callback procedure is run. The other values in the window haven't been updated yet.



Customer ID	ADAMPARK0001
Name	Aaron Fitz Electrical
Short Name	Aaron Fitz Elec
Statement Name	Aaron Fitz Electrical

The Name field still contains the previous value.

You can prevent this problem by registering separate notifications for the Customer ID and Name controls. When each value changes, the callback procedure for that control can then retrieve the correct value.

Working with scrolling windows

To effectively work with scrolling windows in Microsoft Dynamics GP, you need to understand how they operate. Scrolling windows are table-based. Each row in the scrolling window corresponds to one record in a table attached to the scrolling window. Scrolling window events occur when records are read from the attached table and when the user interacts with the scrolling window.

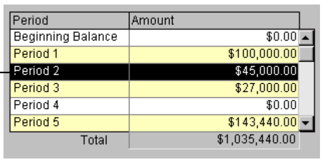
Scrolling window types

There are three types of scrolling windows: browse-only, editable and adds-allowed. Each type has unique characteristics.

Browse-only scrolling windows

A browse-only scrolling window only displays records from its attached table. The user can "browse" through the contents, but can't make changes or add items to the scrolling window.

A browse-only scrolling window allows you to select one item in the grid at a time.

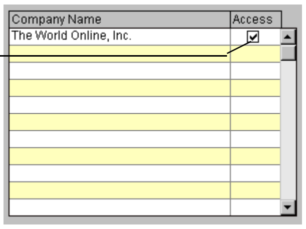


Period	Amount
Beginning Balance	$0.00
Period 1	$100,000.00
Period 2	$45,000.00
Period 3	$27,000.00
Period 4	$0.00
Period 5	$143,440.00
Total	$1,035,440.00

Editable scrolling windows

An editable scrolling window allows the user to change the contents of the selected row. These changes are saved in the table attached to the scrolling window.

You can edit items in an editable scrolling window.

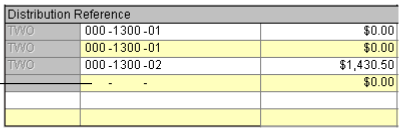


Company Name	Access
The World Online, Inc.	✓

Adds-allowed scrolling windows

An adds-allowed scrolling window has a blank line at the bottom where the user can add new information. The new information is stored in the table attached to the scrolling window.

An adds-allowed scrolling window has a blank line that allows you to add items.



Distribution Reference		
TWO	000 -1300 -01	$0.00
TWO	000 -1300 -01	$0.00
TWO	000 -1300 -02	$1,430.50
	- -	$0.00

Accessing a scrolling window

You can access only the current line in a scrolling window. This means you can set or retrieve the values of fields that appear in the current line. Keep in mind that you won't know which line in the data grid is actually the current line; you must rely upon the data values of the fields in the line to ascertain which line is selected.

Scrolling window events

Your integrating application can be notified when any of the six events for scrolling windows occur. Responding to these events is the basis for integrating with a scrolling window. The remainder of this section describes each scrolling window event and explains how your application can use it.

Data Entry event

Description	Occurs when the user changes the value of any item in the current row of the scrolling window and moves the focus to another row or to another field in the form.
Applies to	Editable and adds-allowed scrolling windows.
Use	Use this event to ascertain when the user has changed the value of any items in the current row of the scrolling window. Use the values of fields in the current row to ascertain which row in the scrolling window was changed.

Delete Row event

Description	Occurs when the user deletes a row from the scrolling window.
Applies to	All scrolling windows.
Use	Use this event to ascertain when the user has deleted a row from the scrolling window. Use the values of the fields in the current row to ascertain which item was deleted.

Got Focus event

Description	Occurs when the focus moves to the scrolling window or moves to a new row in the scrolling window.
Applies to	All scrolling windows.
Use	Use this event to ascertain when the focus has moved to the scrolling window or moved to a different row in the scrolling window. Use the values of the fields in the current row to ascertain which row received the focus.

Insert Row event

Description	Occurs when the user inserts a row into a scrolling window. This event does not occur when the focus moves to a new add-line in an adds-allowed scrolling window.
Applies to	Adds-allowed scrolling windows.
Use	Use this event to ascertain when a new row has been inserted into a scrolling window.

Load Row event

Description

Occurs when a row is read from the attached table and when the focus moves to a new row in the scrolling window. When a scrolling window is initially displayed, the load row event occurs for each record read from the attached table until all lines in the scrolling window are filled. When the focus moves to a new line in the scrolling window, the load row event occurs to refresh the data in the line, then the Got Focus event occurs.

Applies to

All scrolling windows.

Use

Use this event to ascertain what data values have been read to be displayed in the current row. Be sure the notification occurs *after* the Microsoft Dynamics GP code runs so values will have been read for the current line.

Lost Focus event

Description

Occurs when the focus moves to a new row in the scrolling window or moves to another field.

Applies to

All scrolling windows.

Use

Use this event to ascertain when the focus has moved from a row in the scrolling window. Use the values of the fields in the current row to ascertain which row the focus moved from.

Managing cross-thread calls

Beginning with Visual Studio 2005, the .NET framework will automatically check for illegal cross-thread calls in applications. The window controls in standard .NET applications are not thread-safe. For this reason, a common cause of an illegal cross-thread call is when one execution thread creates a control, and another execution thread attempts to update the value of that control. When cross-thread calls are detected, an exception is thrown and the application will not run properly.

Because of the callback architecture used for the Continuum API, you will likely encounter exceptions for illegal cross-thread calls when you run your integration. The issue occurs because one thread created the windows and controls for your integrating application, but the callbacks from the Microsoft Dynamics GP runtime are running on another thread. When the callback code tries to interact with the windows and controls in your application, you will see exceptions when running in the Visual Studio debugger. When running outside of the debugger, you may see incorrect behavior in your integrating application, such as controls that do not update when callbacks occur.

There are two ways that cross-thread issues can be resolved. You can disable the check for cross-thread calls, or you can implement delegates in your integrating application that will perform the actions that are causing the cross-thread issues.

Disabling cross-thread checking

The cross-thread calls that occur when code in your callbacks attempts to interact with the .NET windows and controls in your integration shouldn't cause issues under normal circumstances. While it isn't the best solution, the check for cross-thread calls can be safely disabled for typical Continuum integrations. The following Visual Basic code can be added to the initialization code for a Continuum integration to disable checking for illegal cross-thread calls.

```
System.Windows.Forms.Control.CheckForIllegalCrossThreadCalls = False
```

Using delegates

The preferred way to prevent cross-thread issues that occur when accessing windows and controls from another thread is to use delegates. Delegates allow you to pass execution from the current thread to the thread that can access the windows and controls. In a typical Continuum integration, you will create delegates for each of the callback operations that interact with the windows and controls in your integration.

To create a delegate, use the following procedure.

1. **Decide what action the delegate will perform.**
   The action is often quite basic, such as setting the value of a control in one of the windows for your integration.

2. **Create a subroutine for that action.**
   In the code for the form, add a subroutine that will perform the action. The code should be part of the form, so it has access to the controls on the form. For example, the following subroutine sets the value of the Product text box control for the Name Wizard sample.

```
Public Sub SetProduct(ByVal Product As String)
    Me.Product.Text = Product
End Sub
```

3. **Add a delegate for the action.**
   The delegate defines the signature (parameters) of the subroutine that will be used to perform the action. The signature of the delegate must match the signature of the subroutine that will perform the action. The delegate is added to the code for the form, typically at the beginning of file before the class definition of the form. Delegates often have the term "handler" included in the name.

   Continuing the example, the following delegate was added at the beginning of the WizardWindow.vb file in the Name Wizard sample. Notice how the signature of the delegate matches the signature of the SetProduct subroutine that was added.

   ```
   Delegate Sub ProductHandler(ByVal value As String)
   ```

4. **Replace the direct call with the delegate.**
   In the callback, you will replace the direct call with the code that will use the delegate. For instance, the following code in the callback for the Name Wizard sample directly sets the value of the Product text box control.

   ```
   WizWindow.Product.Text = Product
   ```

   You will replace this code that directly accesses the Product text box control with the delegate that uses the SetProduct subroutine to set the value of the Product text box control. The following code shows how the delegate is used.

   ```
   WizWindow.Invoke(New ProductHandler(AddressOf WizWindow.SetProduct), New
   Object() {Product})
   ```

   There are several parts to the statement:

   - **WizWindow** is the instance of window object that contains the control to be updated.

   - The **Invoke()** method on this window instance is called to activate the delegate. This ensure that the window's thread is used to perform the action. The **Invoke()** method takes two parameters: a delegate instance and set of parameters.

   - At the time the new instance of the **ProductHandler** delegate is created, the method that will perform the operation is specified. In this case, the **SetProduct** method of the WizWindow object will perform the operation for the delegate. The **AddressOf** operator is required by Visual Basic to point to this method instance.

   - The new instance of the ProductHandler delegate also specifies the parameters that will be passed from the delegate to the method that will actually perform the operation. The parameters are passed using a generic object and an array list the specifies the parameters to pass. In this example, **New Object()** creates the parameter object. The parameter values to be passed to the method are enclosed in braces { }. The order of the parameters must match the order of the parameters defined for the method the delegate is calling.

5. **Test the delegate.**
   When the callback is run, the delegate should use the specified method to perform the action. In this example, the Product text control will be updated with the value returned from the callback. When running in the Visual Studio debugger, no cross-thread exceptions should occur for the operation.

User Account Control (UAC)

User Account Control (UAC) in Windows Vista, Windows 7, and Windows Server 2008 can affect how Continuum-based integrations run. To establish a COM connection between the Continuum-based integration and Microsoft Dynamics GP, both applications must be running at the same privilege level. For instance, if Microsoft Dynamics GP is running with elevated privileges, the Continuum integration must also be running with elevated privileges. If the applications are not running with the same privilege level, a new instance of the Microsoft Dynamics GP runtime will be started when the Continuum-based application is launched and attempts to create a COM connection.

You may encounter this issue when you run a Continuum-based integration with the Visual Studio debugger. For improved compatibility with UAC, Visual Studio is often set to run with elevated privileges. If Visual Studio is running with elevated privileges, and a Continuum integration is run with the Visual Studio debugger, the integration will be run with elevated privileges. The integration will be able to create a COM connection only if Microsoft Dynamics GP is running with elevated privileges as well.

Chapter 7: Packaging

Once your integration is complete, you can package the application as you normally would. There are some additional issues you may need to address. Information about packaging your integration is contained in the following sections:

- *Runtime components*
- *Registering Microsoft Dynamics GP as an Automation server*

Runtime components

You may need to include additional runtime components with your Continuum integration. For example, if you create your integration with Visual Basic .NET, you must be sure the appropriate version of the .NET Framework is installed on each Microsoft Dynamics GP workstation that will be using your integration. You may also need to include any "interop" assemblies that are needed for COM components you use for your integration. This includes the **Interop.Dynamics.dll** assembly, which is needed to allow Visual Basic .NET to access the COM API within Microsoft Dynamics GP.

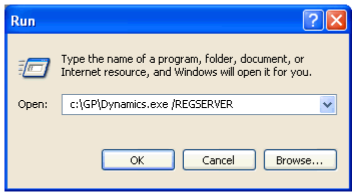
It's important that you test your Continuum integration on a Microsoft Dynamics GP installation that doesn't contain the development environment you used to create the integration. This is the best way to determine whether you must include additional runtime components with your Continuum integration.

Registering Microsoft Dynamics GP as an Automation server

When Microsoft Dynamics GP is installed, the installation routine automatically registers Microsoft Dynamics GP as an Automation server. It will not be necessary to perform this registration, unless the registration information is somehow damaged in the Registry.

If you need to register Microsoft Dynamics GP as an Automation server, start the runtime engine (DYNAMICS.EXE) with the /REGSERVER command line option. Do this only one time, not each time you start the Microsoft Dynamics GP application. Once you register Microsoft Dynamics GP as an Automation server, it remains registered.

One way to register Microsoft Dynamics GP as an Automation server is to chose Run from the Start menu and start the Microsoft Dynamics GP application runtime engine as shown in the following illustration.


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open:
OK
Cancel
Browse...

```
c:\GP\Dynamics.exe /REGSERVER
```

The the Microsoft Dynamics GP application runtime engine will start, add the appropriate information to the Windows Registry and then shut down. No windows will be displayed.

PART 3: INTEGRATION EXAMPLES

Part 3: Integration Examples

This portion of the documentation describes several examples of how the Continuum API can be used to create applications that integrates with Microsoft Dynamics GP. The following applications are discussed:

- Chapter 8, "Field Defaulter," describes an application that automatically defaults information for a field in a Microsoft Dynamics GP window.
- Chapter 9, "Additional Information Window," describes an application that keeps the information in a window synchronized with information in a Microsoft Dynamics GP window.
- Chapter 10, "Name Wizard," describes an application that uses the Continuum API to look up the names of resources in Microsoft Dynamics GP.

Chapter 8: Field Defaulter

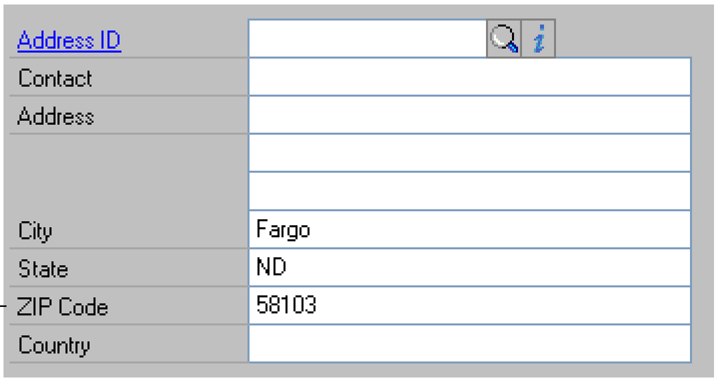
This sample Visual Basic .NET application will automatically enter the correct city and state values based upon the ZIP code entered in the Microsoft Dynamics GP Customer Maintenance window. The following topics are discussed:

- *Overview*
- *Running the sample application*
- *How the Continuum API was used*

Overview

This sample Visual Basic application has an extremely simple a user interface. A single window is displayed to tell you the integration is running. As the application runs in the background, it monitors the ZIP Code field in the Microsoft Dynamics GP Customer Maintenance window. When the ZIP Code changes, the Visual Basic application attempts to fill in the correct City and State.

When you enter the ZIP Code, the values for the City and State are entered automatically.



Address ID	
Contact	
Address	
City	Fargo
State	ND
ZIP Code	58103
Country	

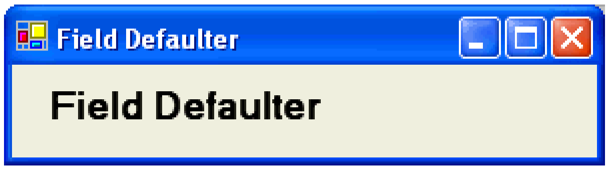
Running the sample application

To run this sample application, perform the following steps.

1. **Start Microsoft Dynamics GP.**
   Microsoft Dynamics GP must be running before you start the sample application. Refer to Chapter 6, "Programming Techniques," to learn more about starting applications that integrate with Microsoft Dynamics GP.

2. **Start Visual Studio .NET and open the solution file for the sample application.**
   The solution file for this sample is named FLDDFLTR.SLN and is located in the Field Default folder inside the Samples folder.

3. **Choose Start from the Debug menu.**
   The solution will be built. If there are no build errors, the following window will be displayed.



4. **Open the Customer Maintenance window in Microsoft Dynamics GP.**

5. **Enter a ZIP Code.**
   For demonstration purposes, this sample application recognizes only a small number of ZIP codes. You can enter any of the follow ZIP codes and have the corresponding City and State filled in:

02109	58104
53151	58474
55111	60605
56560	85012
58078	95014
58102	98052
58103	

   When you have finished working with the sample application, close Microsoft Dynamics GP. The sample application will close automatically and return you to Visual Studio development mode.

How the Continuum API was used

This sample application uses three methods and two triggers from the Continuum API. The declarations for the initialization code are contained in the TemplateMain.vb code module, and the actual Continuum initialization is found in the Load() method for the Field Defaulter window.

Methods

This Visual Basic .NET application uses the **GetDataValue method** to retrieve the ZIP Code value. In the calback for each focus trigger the **SetDataValue method** is used to set the values of the City and State in the Customer Maintenance window, based upon the ZIP Code value retrieved.

Triggers

This sample application uses two triggers. The first trigger is registered for the Toolbar form. The Toolbar form closes when the Microsoft Dynamics GP application is closed. When this occurs, the corresponding method in the callback class disposes of the Microsoft Dynamics GP application object and closes the Visual Basic application.

The second trigger watches the ZIP Code in the Microsoft Dynamics GP Customer Maintenance window. When the ZIP Code value changes, the corresponding procedure in the callback class retrieves the value and attempts to look up the city and state that correspond to the ZIP Code.

When you examine the source for this sample integration, you will see that it contains wrappers many of the Continuum triggers that you can use. You may want to use similar wrappers in your integration to make adding triggers easier.

Chapter 9: Additional Information Window

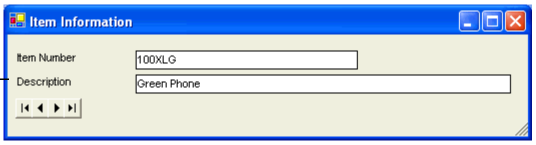
This sample Visual Basic .NET application shows how your can use the Continuum API to track additional information in an integrating application. The following topics are discussed:

- *Overview*
- *Running the sample application*
- *How the Continuum API was used*

Overview

This sample Visual Basic application consists of an Item Information window that shows how you would track additional information for an inventory item. The triggers and methods provided by the Continuum API are used to keep the information in the Item Information window synchronized with the information displayed in the Item Maintenance window.

The information displayed here is kept synchronized with the Item Maintenance window.



Item Number: 100XLG
Description: Green Phone

Running the sample application

To run this sample application, perform the following steps.

1. **Start Microsoft Dynamics GP.**
   Microsoft Dynamics GP must be running before you start the sample application. Refer to Chapter 6, "Programming Techniques," to learn more about starting applications that integrate with Microsoft Dynamics GP.

2. **Start Visual Studio .NET and open the solution file for the sample application.**
   The solution file for this sample is named ADDLINFO.SLN and is located in the Additional Information folder inside the Samples folder.

3. **Choose Start from the Debug menu.**
   The solution will be built. If there are no build errors, the Item Information window will be displayed.

4. **Open the Item Maintenance window in Microsoft Dynamics GP.**

5. **Experiment with the Item Maintenance and Item Information windows.**
   You should be able to retrieve items, clear the window and use the browse buttons in the Item Maintenance window. The information in the Item Information window should remain synchronized with the information in the Item Maintenance window in Microsoft Dynamics GP. You can also click the browse buttons in the Item Image window.

   When you have finished working with the sample application, close Microsoft Dynamics GP. The sample application will close automatically and return you to Visual Studio development mode.

How the Continuum API was used

This sample application uses several methods and triggers from the Continuum API. The declarations for the initialization code are contained in the AddlInfo.vb code module, and the actual Continuum initialization is found in the Load() method for the Item Information window.

Methods

Each of the browse buttons in the Item Information window uses the **MoveToField method** and the **ExecuteSanScript method** to push the corresponding browse button in the Microsoft Dynamics GP Item Maintenance window. The **GetDataValue method** is used in the callback for the triggers that are activated when the Item Number or Description have changed. These methods retrieve the values of the Item Number and Description, and copy them to the corresponding fields in the Item Information window.

Triggers

This sample application uses five triggers. The first trigger adds the Item Image menu item to the Extras menu for the Microsoft Dynamics GP Item Maintenance window. When the menu item is chosen, the corresponding procedure in the callback class displays the Item Information window in the integrating application.

The second trigger is registered for the Item Maintenance window. When this window closes, the corresponding procedure in the callback class causes the Item Image window to be minimized.

The third trigger is registered for the Toolbar form. The Toolbar form closes when the Microsoft Dynamics GP application is closed. When this occurs, the corresponding procedure in the callback class disposes of the Microsoft Dynamics GP application object and closes the Visual Basic application.

The last two triggers watch the Item Number and Description in the Microsoft Dynamics GP Item Maintenance window. When the content of these items change, the corresponding procedures in the callback class use Continuum methods to retrieve the Item Number and Description.

When you examine the source for this sample integration, you will see that it contains wrappers many of the Continuum triggers that you can use. You may want to use similar wrappers in your integration to make adding triggers easier.

Chapter 10: Name Wizard

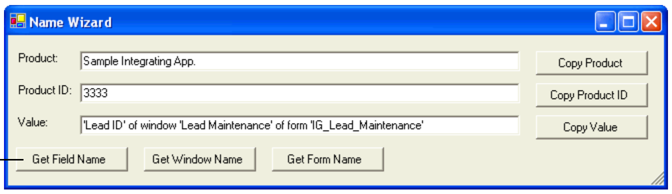
This sample Visual Basic .NET application uses methods from the Continuum API to retrieve resource names from the Microsoft Dynamics GP application. You will find this sample to be a useful tool while developing your Continuum integrations. The following topics are discussed:

- *Overview*
- *Running the sample application*
- *How the Continuum API was used*

Overview

This sample Visual Basic application consists of single window that allows you to look up the name of a form, window, or field in Microsoft Dynamics GP simply by clicking on it in the application.

Click the type of name you want to retrieve.



Product: Sample Integrating App.
Product ID: 3333
Value: 'Lead ID' of window 'Lead Maintenance' of form 'IG_Lead_Maintenance'

Click one of the buttons at the bottom of the window to specify what type of name you want to retrieve. You will be placed into "wizard mode" in the Microsoft Dynamics GP application. Click on the resource for which you want to retrieve the name. The product, product ID, and fully-qualified name will be displayed in the Name Wizard. Use the buttons on the right side of the window to copy the content to the clipboard so you can paste it into your Continuum integration code.

Running the sample application

To run this sample application, perform the following steps.

1. **Start Microsoft Dynamics GP.**
   Microsoft Dynamics GP must be running before you start the sample application.

2. **Start Visual Studio .NET and open the solution file for the sample application.**
   The solution file for this sample is named NAMEWIZARD.SLN and is located in the Name Wizard folder inside the Samples folder.

3. **Choose Start from the Debug menu.**
   The solution will be built. If there are no build errors, the Name Wizard window will be displayed.

4. **Open a window in Microsoft Dynamics GP.**
   You must open the window in Microsoft Dynamics GP from which you want to retrieve name information before you start wizard mode.

5. **Select the type of name you want to retrieve.**
   In the Name Wizard, click one of the buttons at the bottom to specify which type of name you want to retrieve. This will start the wizard mode and activate the Microsoft Dynamics GP application. The pointer will change to indicate that you are in wizard mode.

6. **Click on an item in Microsoft Dynamics GP.**
   While in wizard mode click on the item for which you want to retrieve the name. The name information will be retrieved and displayed in the Name Wizard.

7. **Use the name information retrieved.**
   Use the buttons on the right side of the Name Wizard to copy the name content to the clipboard so you can paste the information into your integration code.

How the Continuum API was used

This sample application uses several methods from the Continuum API. The declarations and Continuum initialization code are contained in the TemplateMain.vb code module.

Methods

The Name Wizard starts and stops the wizard mode using the StartWizardMode method and StopWizardMode method. These methods use a callback in the GPCallback class to return name and product information from the item selected. The ExecuteSanScript method is used to execute pass-through sanScript to retrieve the product ID for the application from which name information was retrieved.

Parameter handler

The Name Wizard sample uses a parameter handler to pass information from the pass-through sanScript to the Visual Basic code. The initialization code for the sample uses the SetParamHandler method to specify that the GPCallback object will be used as the parameter handler. A string variable named ProdID is defined in this class. The **OLE_SetProperty()** function is used from pass-through sanScript to set the value of the ProdID variable, allowing the Visual Basic code to access the value and display it in the Name Wizard window.

Part 4: The Continuum Integration Library

This part describes the objects that Microsoft Dynamics GP makes available to other applications through the Continuum API. It provides detailed descriptions of the properties and methods for each object. The information is divided into the following areas.

- Chapter 11, "Application Object," describes the properties and methods for the Application object.
- Chapter 12, "Field Object," describes the properties of the Field object.

The syntax descriptions and programming style used in this part are similar to those used in Visual Basic's documentation.

Chapter 11: Application Object

Your integrating application will use the Application object to perform many tasks in Microsoft Dynamics GP. The methods and properties that apply to the Application object are listed below. A detailed explanation of each appears on the following pages:

- Activate method
- CallVBAMacro method
- CreateFieldObject method
- CurrentProduct property
- CurrentProductID property
- ExecuteSanScript method
- ExternalProductName property
- GetDataValue method
- GetDataValueEx method
- GetVBAGlobals method
- Hide method
- MacroError property
- MoveToField method
- RegisterDatabaseTrigger method
- RegisterFocusTrigger method
- RegisterFormTrigger method
- RegisterShutdownRoutine method
- RegisterWatchTrigger method
- SetDataValue method
- SetDataValueEx method
- SetParamHandler method
- Show method
- StartWizardMode method
- StopProcessing method
- StopWizardMode method
- VBE method
- Visible property

Activate method

Description Causes the application to become the active application.

Syntax *object*.**Activate**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Return value None

Comments This method is typically used to bring the Microsoft Dynamics GP application to the foreground.

Examples This example uses the **Activate method** to bring the Microsoft Dynamics GP application to the foreground.

```
GPApp.Activate
```

CallVBAMacro method

Description

Executes the specified procedure within the VBA environment.

Syntax

object.**CallVBAMacro(***procedure_name***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *procedure_name* – A string containing the name of the procedure to be run in the VBA environment. The procedure to run must not have any parameters. When specifying the procedure to run, use the form:

  project_name.[*module_name*].*procedure_name*.

  The procedure name does not have to be fully qualified if it is unique within the current VBA project.

Return value

An integer. The value 0 indicates the procedure couldn't be found and executed.

Examples

This example uses the **CallVBAMacro method** to execute the ShowName procedure in the TimeBilling module for the Dynamics_GP project.

```
Private Sub CallVBAMacro_Click()
    GPApp.CallVBAMacro("Dynamics_GP.[TimeBilling].ShowName")
End Sub
```

CreateFieldObject method

Description

Creates a field object for the specified field in the application.

Syntax

object.**CreateFieldObject(***field_name***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the field for which a field object will be created. The field name must be fully qualified.

Return value

A **field** object

Comments

Use the properties of the **field** object to ascertain characteristics of the field.

Examples

This example uses the **CreateFieldObject method** to create a field object for the Customer Number field in the Microsoft Dynamics GP Customer Maintenance window.

```
Private Sub CreateFieldObject_Click()
    Dim FieldObj As Object
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
End Sub
```

Related items

Additional information
Chapter 12, "Field Object"

CurrentProduct property

Description

Returns a string containing the name of the currently-active product, or sets the currently-active product for the application.

Syntax

object.**CurrentProduct** [= *product_name*]

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *product_name* – A string containing the name of the product that will be made the currently-active product. This name must be the same as the product name that appears in the Microsoft Dynamics GP launch file.

Comments

This property is used to switch between products when Microsoft Dynamics GP is operating in a multidictionary configuration. You must make a product the current product before you can perform other operations in it.

Since the name of a product can change, you should consider using the CurrentProductID property to specify the current product.

Examples

This example uses the **CurrentProduct property** to retrieve the name of the currently-active product in the Microsoft Dynamics GP application.

```
Private Sub CurrentProduct_Click()
    Dim CurrentProd As String
    CurrentProd = GPApp.CurrentProduct
End Sub
```

This example uses the **CurrentProduct property** to make the Sample Integrating App. product the current product in the Microsoft Dynamics GP application.

```
Private Sub CurrentProduct_Click()
    GPApp.CurrentProduct = "Sample Integrating App."
End Sub
```

Related items

Commands
CurrentProductID property

CurrentProductID property

Description

Returns an integer containing the product ID of the currently-active product, or sets the currently-active product for the application.

Syntax

object.**CurrentProduct** [= *product_ID*]

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *product_ID* – An integer containing the product ID of the product that will be made the currently-active product. This name must be the product ID of a product that appears in the Microsoft Dynamics GP launch file.

Comments

This property is used to switch between products when Microsoft Dynamics GP is operating in a multidictionary configuration. You must make a product the current product before you can perform other operations in it.

The **CurrentProductID property** is the preferred method for specifying the current product, because unlike the product name, the product ID is unlikely to change for future releases.

Examples

This example uses the **CurrentProductID property** to retrieve the product ID of the currently-active product in the Microsoft Dynamics GP application.

```
Private Sub CurrentProduct_Click()
    Dim CurrentProdID As Integer
    CurrentProdID = GPApp.CurrentProductID
End Sub
```

This example uses the **CurrentProductID property** to make the Sample Integrating App. product (product ID 3333) the current product in the Microsoft Dynamics GP application.

```
Private Sub CurrentProduct_Click()
    GPApp.CurrentProductID = 3333
End Sub
```

Related items

Commands
CurrentProduct property

ExecuteSanScript method

Description

Sends sanScript code into the application to be compiled and executed.

Syntax

object.**ExecuteSanScript(***code_string*, *compile_error_message***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *code_string* – A string containing the sanScript code to be compiled and executed.
- *compile_error_message* – A returned string containing any compiler error that occurred when the sanScript code was compiled.

Return value

An integer indicating the compiling status. The value 0 indicates the sanScript code compiled successfully. Any other value indicates a compiler error occurred.

Examples

This example uses the **ExecuteSanScript method** to execute sanScript code to open the Customer Maintenance window.

```
Private Sub ExecuteSanScript_Click()
    Dim Err_Num As Integer
    Dim ErrorMsg As String
    Err_Num = GPApp.ExecuteSanScript( _
    "open form RM_Customer_Maintenance;", ErrorMsg)

    If Err_Num <> 0 Then
        'A compiler error occurred. Display the error.
        MsgBox ErrorMsg
    End If
End Sub
```

Related items

Additional information
Chapter 3, "Pass-through sanScript"

ExternalProductName property

Description

Sets or returns a string that specifies the name of the application that is integrating with Microsoft Dynamics GP through the Continuum API.

Syntax

object.**ExternalProductName** [=*string*]

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *string* – A string containing the name of the product that will be integrating with Microsoft Dynamics GP through the Continuum API.

Comments

We recommend that you set this property immediately after you create an instance of the Microsoft Dynamics GP application object in your integrating application's code. You should set the **ExternalProductName property** before you register any triggers.

Examples

This example uses the **ExternalProductName property** to tell Microsoft Dynamics GP the name of the application that is integrating with Microsoft Dynamics GP.

```
Private Sub SetExternalProductName()
    GPApp.ExternalProductName = "Quick GL Entry"
End Sub
```

GetDataValue method

Description Retrieves the data value from the specified field.

Syntax *object*.**GetDataValue(***field_name***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the field for which the data value will be retrieved. The field name must be fully qualified.

Return value A string containing the data value.

Comments The following table lists each control type for which a data value can be returned, as well as a description of the value returned from it.

Control type	Description
Button drop list	A numeric value that identifies the item last selected in the list.
Check box	The value 0 if the check box is not marked, 1 if the check box is marked.
Combo box	The text of the item selected in the combo box.
Composite	The data in the composite field, including any formatting.
Currency	The value in the currency field, including any formatting.
Date	The value in the date field, including any formatting.
Drop-down list	A numeric value that identifies the item selected in the list.
Integer	The numeric value in the field.
List box	A numeric value that identifies the item selected in the list.
Long integer	The numeric value in the field.
Multi-select list box	A 32-bit numeric value that identifies which items in the list are marked.
Progress indicator	The numeric value in the field.
Radio group	A numeric value that identifies which radio button is selected in the group.
String	The string value in the field, including any formatting.
Time	The value in the time field, including any formatting.
Visual switch	A numeric value that identifies the item selected in the visual switch.

If you are using **GetDataValue** in the callback method for a database trigger, you don't need to fully qualify the location of the table buffer you're accessing. By default, the **GetDataValue method** will access the table buffer for the table that caused the database trigger to run.

Examples This example uses the **GetDataValue method** to retrieve the value in the Customer Number field of the Customer Maintenance window.

```
Private Sub GetDataValue_Click()
    Dim CustNumber As String
    CustNumber = GPApp.GetDataValue( _
    "'Customer Number' of window RM_Customer_Maintenance" & _
    "of form RM_Customer_Maintenance")
End Sub
```

This example uses the **GetDataValue method** to retrieve the value in the Customer Number field of the RM_Customer_MSTR table that is used for the RM_Customer_Maintenance form.

```
Private Sub GetDataValue_Click()
    Dim CustNumber As String
    CustNumber = GPApp.GetDataValue( _
    "'Customer Number' of table RM_Customer_MSTR of form " & _
    "RM_Customer_Maintenance")
End Sub
```

Related items

Commands

GetDataValueEx method, **SetDataValue method**, **SetDataValueEx method**

GetDataValueEx method

Description

Retrieves the data value from the specified currency field and applies the designated format.

Syntax

object.**GetDataValueEx(***field_name*, *format_selector***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the currency field for which the data value will be retrieved.
- *format_selector* – An integer indicating which format string to apply to the value returned from the field. The following table lists the integer values and the corresponding format that will be applied.

Integer value	Format
0	Control Panel Defaults
1	1,234.
2	1,234.5
3	1,234.56
4	1,234.567
5	1,234.5678
6	1,234.56789
7	$1,234.
8	$1,234.5
9	$1,234.56
10	$1,234.567
11	$1,234.5678
12	$1,234.56789

Return value

A string containing the currency value with the designated format applied.

Examples

This example uses the **GetDataValueEx method** to retrieve the value in the Sales Amount currency field in the Receivables Transaction Entry window. The value will be returned with two decimal places, but won't display the currency symbol.

```
Private Sub GetDataValueEx_Click()
    Dim SalesAmount As String

    SalesAmount = GPApp.GetDataValueEx( _
    "'Sales Amount' of window RM_Sales_Entry" & _
    "of form RM_Sales_Entry", 3)
End Sub
```

Related items

Commands

GetDataValue method, SetDataValue method, SetDataValueEx method

GetVBAGlobals method

Description

Retrieves a reference to the globals in the Visual Basic for Applications (VBA) environment embedded in Microsoft Dynamics GP. This provides access to items in the DUOS (Dynamic User Object Store) that is part of the Microsoft Dynamics GP VBA implementation. The VBA environment must be open and active in Microsoft Dynamics GP for this method to work properly.

Syntax

object.**GetVBAGlobals**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Return value

A reference to the globals of the Visual Basic for Applications environment.

Comments

Refer to the VBA Developer's Guide for more information about accessing items from the DUOS.

Examples

The following example uses the **GetVBAGlobals method** to retrieve a reference to the globals for the Visual Basic for Applications environment within Microsoft Dynamics GP. Then the DUOSObjectCombineID method is accessed through the globals reference to create a data object ID.

```
Public Sub GetVBAGlobals_Click()
    Dim VBAGlobals As Object
    Dim objID As String
    Set VBAGlobals = GPApp.GetVBAGlobals
    objID = VBAGlobals.DUOSObjectCombineID("Microsoft Dynamics GP", "A")
End Sub
```

Hide method

Description

Causes Microsoft Dynamics GP to become hidden.

Syntax

object.**Hide**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Return value

None

Examples

The following example uses the **Hide method** to make the Microsoft Dynamics GP application invisible.

```
Public Sub HideDynamics_Click()
    GPApp.Hide
End Sub
```

Related items

Commands

Show method

MacroError property

Description

Returns an integer that describes the result of the **MoveToField method**.

Syntax

object.**MacroError**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Comments

Use the **MacroError property** after you execute the **MoveToField method** to ascertain the results of the move. The following table lists the possible values that can be returned, along with a description of each.

Value	Description
0	No error occurred. The **MoveToField method** was successful.
1	The window containing the field is not open or does not exist.
2	The focus was diverted by a **focus field** statement in sanScript code.
3	A **restart field** statement in sanScript code was encountered. The focus was not moved.
4	Attempted to move to an unfocusable field.

Examples

This example uses the **MacroError property** to retrieve the result of the **MoveToField method**.

```
Private Sub MacroError_Click()
    GPApp.MoveToField("'Customer Number' of window " & _
    "RM_Customer_Address of form RM_Customer_Address")
    If GPApp.MacroError <> 0 Then
        MsgBox "The focus couldn’t be moved to the Customer ID."
    End If
End Sub
```

Related items

Commands

MoveToField method

MoveToField method

Description

Moves the focus to the specified field in the application.

Syntax

object.**MoveToField(***field_name***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the field to which the focus will be moved. The field name must be fully qualified.

Return value

An integer indicating the result of the move. The value 0 indicates the focus did not move to the specified field. The value 1 indicates the focus did move to the specified field.

Examples

This example uses the **MoveToField method** to move the focus to the Customer Number field in the Microsoft Dynamics GP Customer Maintenance window.

```
Private Sub CreateFieldObject_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.MoveToField("'Customer Number' " & _
    "of window RM_Customer_Maintenance of form " & _
    "RM_Customer_Maintenance")
    If ErrVal <> 1 Then
        MsgBox "Unable to move focus to the Customer ID field."
    End If
End Sub
```

Related items

Commands

MacroError property

RegisterDatabaseTrigger method

Description

Registers a database trigger for the Microsoft Dynamics GP application. Database triggers respond to *successful* table operations in an application, such as saving a record, deleting a record or reading a record.

Syntax

object.**RegisterDatabaseTrigger(***table_name*, *form_name*, *table_operations*, *callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *table_name* – A string containing the name of the table for which the database trigger is being registered. The string must include the qualifier “table”.
- *form_name* – A string containing the name of the form to which the database trigger will be restricted. The string must include the qualifier “form”. The trigger will run for the database operations originating from this form only. It won’t run for table operations originating from other areas. If you don’t want to restrict the trigger to a particular form, set this parameter to the empty string "".
- *table_operations* – An integer that specifies which table operations cause the trigger to run. The following table lists the table operations and their corresponding integer values. You can add these values together to run a trigger for more than one table operation. For instance, the integer “3” runs a database trigger for all types of database read operations.

Operation	Value	Description
Read without lock	1	Occurs when Microsoft Dynamics GP reads a record in the table without locking it.
Read with lock	2	Occurs when Microsoft Dynamics GP reads a record in the table with either a passive or active lock.
Add	4	Occurs when Microsoft Dynamics GP adds a new record to the table.
Update	8	Occurs when Microsoft Dynamics GP updates a record in the table.
Delete	16	Occurs when Microsoft Dynamics GP deletes a record in the table.

- *callback_object* – The name of the callback object in the integrating application containing the method to be run in response to the database trigger.
- *callback_method* – A string containing the name of the method in the callback object of the integrating application. This method will run in response to the database trigger.

Return value

An integer indicating whether the trigger was registered properly. The following table lists the possible values that can be returned:

Value	Description
0	No error occurred.
1	An unknown error occurred and the trigger was not registered.
4	The table or form could not be found.

Comments

In the callback method for the database trigger, you can use the **GetDataValue method** to retrieve field values from the table buffer. When you do this, you don't need to fully qualify the location of the table buffer you're accessing. By default, the **GetDataValue method** will access the table buffer for the table that caused the database trigger to activate.

Examples

This example uses the **RegisterDatabaseTrigger method** to register a database trigger for delete operations that occur in the RM_Customer_MSTR table. The RMCustDel procedure in the GPCallback class will be run when the trigger runs.

```
Private Sub RegisterDatabaseTrigger_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterDatabaseTrigger( _
    "table RM_Customer_MSTR", "", 16, GPCallback, "RMCustDel")
    If ErrVal <> 0 Then
        MsgBox "Unable to register the database trigger."
    End If
End Sub
```

This example registers a database trigger for any read operations of the RM_Customer_MSTR table that originate from the Customer Maintenance form.

```
Private Sub RegisterDatabaseTrigger_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterDatabaseTrigger( _
    "table RM_Customer_MSTR", "form RM_Customer_Maintenance", _
    3, GPCallback, "RMCustRead")
    If ErrVal <> 0 Then
        MsgBox "Unable to register the database trigger."
    End If
End Sub
```

RegisterFocusTrigger method

Description

Registers a focus trigger for the Microsoft Dynamics GP application. Focus triggers respond to "focus" events in an application, such as a window opening or closing, or the focus moving from one field to the next.

Syntax

object.**RegisterFocusTrigger(***qualified_resource*, *focus_type*, *attach_type*, *callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *qualified_resource* – A string containing qualified name of the resource for which the focus trigger is being registered. This parameter will have a form such as:
  - form *form_name*
  - window *window_name* of form *form_name*
  - window *scrolling_window_name* of form *form_name*
  - *field_name* of window *window_name* of form *form_name*
- *focus_type* – An integer that identifies which focus event causes the trigger to run. The following table lists the focus events and the resources to which they apply:

Event	Value	Resources
PRE	0	Fields, windows, forms and scrolling windows
CHANGE	1	Fields and scrolling windows
POST	2	Fields, windows, forms and scrolling windows
PRINT	3	Windows
ACTIVATE	4	Windows
FILL	5	Scrolling windows
INSERT	6	Scrolling windows
DELETE	7	Scrolling windows
MODAL DIALOG	8	Modal dialog (dialogs generated by **error**, **warning**, **ask()** or **getstring()** sanScript commands)
CONTEXT MENU	9	Context menu

- *attach_type* – An integer indicating when the focus trigger runs relative to the original focus event:

Value	Description
1	Trigger runs before the focus event.
2	Trigger runs after the focus event.

- *callback_object* – The name of the callback object in the integrating application containing the method to be run in response to the focus trigger.
- *callback_method* – A string containing the name of a method in the callback object for the integrating application. This method will run in response to the focus trigger.

Return value

An integer indicating whether the trigger was registered properly. The following table lists the possible values that can be returned:

Value	Description
0	No error occurred.
1	An unknown error occurred and the trigger was not registered.
4	The specified resource could not be found.

Comments

For modal dialog triggers, the callback method must use the following format for its definition:

MethodName (*dialog_type*, *prompt*, *control_1*, *control_2*, *control_3*, *answer*)

MethodName – The name of the callback method.

dialog_type – An integer specifying the type of dialog being displayed.

Value	Description
0	Indicates an **error**, **warning**, **ask()** dialog is being displayed.
1	Indicates a **getstring()** dialog is being displayed.

prompt – A string containing the text displayed in the dialog. When checking the value of this parameter, be sure you have the capitalization and spelling correct.

control_1 – A string containing the text displayed in button 1 of the dialog.

control_2 – A string containing the text displayed in button 2 of the dialog.

control_3 – A string containing the text displayed in button 3 of the **ask()** dialog. For **getstring()** dialogs, this is the editable string displayed.

answer – An integer specifying what button was pressed (after event) or is to be pressed (before event). The value will depend on the type of dialog that was displayed.

Dialog type	Value	Description
error or **warning**	0	Indicates the OK button was pressed.
ask()	0	Indicates that button 1 was pressed.
	1	Indicates that button 2 was pressed.
	2	Indicates that button 3 was pressed.
getstring()	-1	Indicates that no button was pressed.
	0	Indicates that the OK button was pressed.
	1	Indicates that the Cancel button was pressed.

For modal dialog triggers, the “before” trigger callback can modify the text of the dialog and the buttons displayed. It can also automatically respond to the dialog by changing the value of the *answer* callback parameter.

Examples

This example uses the **RegisterFocusTrigger method** to register a focus trigger for the Change event of the Clear button in the Customer Maintenance window. The CustMaintClear method in the GPCallback class is called when the trigger runs.

```
Private Sub RegisterFocusTrigger_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterFocusTrigger( _
    "'Clear Button' of window RM_Customer_Maintenance of " & _
    "form RM_Customer_Maintenance", 1, 2, GPCallback, _
    "CustMaintClear")
    If ErrVal <> 0 Then
        MsgBox "Unable to register the focus trigger."
    End If
End Sub
```

This example uses a modal dialog trigger to handle the **ask()** dialog that is displayed in the Sales Transaction Entry window when a Customer ID is entered, but the customer has not been defined. The trigger is registered to run before the dialog is displayed. In the callback for the trigger, the content of the dialog is examined, a message is displayed, and the dialog is dismissed without being shown to the user.

The following is the registration for the modal dialog trigger.

```
GPApp.RegisterFocusTrigger("window 'SOP_Entry' of form 'SOP_Entry'", 8, 1,
GPCallback, "cbSOPCustomerID")
```

The following is the callback method that is run in response to the trigger. Notice that the parameters that can be set by the callback are passed by reference and not by value.

```
Public Sub cbSOPCustomerID(ByVal DialogType As Integer, ByRef Prompt As
String, ByRef Control1 As String, ByRef Control2 As String, ByRef Control3 As
String, ByRef Answer As Integer)
    Dim sanScript As String
    Dim err_val As Integer
    Dim err_msg As String

    'Verify that it is an ask() dialog
    If DialogType <> 0 Then
        Exit Sub
    Else
        'Check the message to verify it is the dialog we expect
        If Prompt = "Do you want to add this customer record?" Then
            sanScript = "warning" + Chr(34) + "This customer does " & _
            "not exist. Please add the customer using the Customer " & _
            "Maintenance window." + Chr(34) + ";"
            err_val = GPApp.ExecuteSanscript(sanScript, err_msg)
            If err_val <> 0 Then
                MsgBox(err_msg)
            End If

            'Dismiss the ask() dialog by clicking button 2
            Answer = 1
            GPApp.StopProcessing()
        End If
    End If
End Sub
```

Related items

Commands

StopProcessing method

RegisterFormTrigger method

Description

Registers a form trigger for the Microsoft Dynamics GP application. If successfully registered, an item will appear in the "Extras" menu when the form for which the trigger was registered is open.

Syntax

object.**RegisterFormTrigger(***form_name*, *menu_item_name*, *accelerator_key*, *callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *form_name* – A string containing the name of the form for which the form trigger is being registered. The string must include the qualifier "form".
- *menu_item_name* – A string containing the name of the menu item that will be added to the Extras menu.
- *accelerator_key* – A string containing the character that will be used as the accelerator key for the new menu item. Be sure this accelerator key does not conflict with any existing accelerator keys. If you don't want an accelerator key, use the empty string ("").
- *callback_object* – The name of the callback object in the integrating application containing the method to be run in response to the form trigger.
- *callback_method* – A string containing the name of a method in the callback object for the integrating application. This method will be run in response to the form trigger.

Return value

An integer indicating whether the trigger was registered properly. The following table lists the possible values that can be returned:

Value	Description
0	No error occurred.
1	An unknown error occurred and the trigger was not registered.
4	The specified form could not be found.

Examples

This example uses the **RegisterFormTrigger method** to add the "Picture" item to the Extras menu that will appear when the Customer Maintenance window is open. When the user chooses Picture from the Extras menu, the CustPicture procedure in the GPCallback class will run.

```
Private Sub RegisterFormTrigger_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterFormTrigger( _
    "form RM_Customer_Maintenance", "Picture", "P", GPCallback, _
    "CustPicture")
    If ErrVal <> 0 Then
        MsgBox "Unable to register the form trigger."
    End If
End Sub
```

RegisterShutdownRoutine method

Description

Specifies the routine in the integrating application that will run when Microsoft Dynamics GP is shut down. This event does *not* occur when you enter the Modifier or the Report Writer.

Syntax

object.**RegisterShutdownRoutine(***callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *callback_object* – The name of the callback object in the integrating application containing the method to be run when Microsoft Dynamics GP shuts down.
- *callback_method* – A string containing the name of a method in the callback object for the integrating application. This method will run when Microsoft Dynamics GP shuts down.

Return value

An integer. The value 0 indicates the shutdown routine was registered properly. Any other value indicates the shutdown routine was not registered.

Comments

The procedure run by this method is typically used to shut down the integrating application.

If you will be using the Modifier or the Report Writer, we recommend that you register a focus trigger for the close event on the Toolbar form in Microsoft Dynamics GP, rather than using the ***RegisterShutdownRoutine method****. The close event on the Toolbar form occurs when you enter the Report Writer or the Modifier, as well as when you shut down the Microsoft Dynamics GP application.*

Examples

This example uses the **RegisterShutdownRoutine method** to cause the Shutdown procedure in the GPCallback class to run when Microsoft Dynamics GP is shut down.

```
Private Sub RegisterShutdownRoutine_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterShutdownRoutine(GPCallback, _
    "Shutdown")
    If ErrVal <> 0 Then
        MsgBox "Error registering shutdown routine."
    End If
End Sub
```

RegisterWatchTrigger method

Description

Registers a watch trigger for a field in the Microsoft Dynamics GP application. Watch triggers run each time the content of the field changes, regardless of where the focus is in the window.

Syntax

object.**RegisterWatchTrigger(***field_name*, *callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the field for which the watch trigger is being registered. The field name must be fully qualified.
- *callback_object* – The name of the callback object in the integrating application containing the method to be run in response to the watch trigger.
- *callback_method* – A string containing the name of a method in the callback object for the integrating application. This method will run in response to the watch trigger.

Examples

This example uses the **RegisterWatchTrigger method** to register a watch trigger for the Customer Number field in the Customer Maintenance window. The CustMaintCustNum procedure in the GPCallback class will run when the trigger runs.

```
Private Sub RegisterWatchTrigger_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.RegisterWatchTrigger( _
    "'Customer Number' of window RM_Customer_Maintenance of " & _
    "form RM_Customer_Maintenance", GPCallback, "CustMaintCustNum")
    If ErrVal <> 0 Then
        MsgBox "Unable to register the watch trigger."
    End If
End Sub
```

SetDataValue method

Description Sets the data value in the specified field.

Syntax *object*.**SetDataValue(***field_name*, *string_value***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the field for which you want to set the data value. The field name must be fully qualified.
- *string_value* – A string containing the value to which you want to set the field.

Return value An integer indicating whether the field value was set. The value 0 indicates the field value was set. Any other value indicates the field value was not set.

Comments The following table lists each control type for which a value can be set, as well as a description of the string used to set the data value.

Control type	Description
Button drop list	A numeric value that identifies the item to select in the list.
Check box	The value 0 to unmark the check box. The value 1 to mark the check box.
Combo box	The text of the item to select in the combo box.
Composite	The value in the composite field, including any formatting.
Currency	The value in the currency field, including any formatting.
Date	The value in the date field, including any formatting.
Drop-down list	A numeric value that identifies the item to select in the list.
Integer	The numeric value for the field.
List box	A numeric value that identifies the item to select in the list.
Long integer	The numeric value for the field.
Multi-select list box	A 32-bit numeric value that identifies which items in the list to mark.
Progress indicator	The numeric value for the field.
Radio group	A numeric value that identifies which radio button to select in the group.
String	The string value for the field, including any formatting.
Time	The value for the time field, including any formatting.
Visual switch	A numeric value that identifies the item to select in the visual switch.

If the value you're setting is negative, you must *precede it with a minus sign (-) to indicate that it is negative, regardless of how the value is displayed in the field.*

You may want to use the **MoveToField method** to move the focus to the destination field before you set the field's value. That way, any validation code for the field will be run when the focus leaves the destination field.

Examples

This example uses the **SetDataValue method** to set the value in the Customer Number field of the Customer Maintenance window.

```
Private Sub SetDataValue_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.SetDataValue( _
    "'Customer Number' of window RM_Customer_Maintenance" & _
    "of form RM_Customer_Maintenance", "ADVANCED0001")
End Sub
```

Related items

Commands

GetDataValue method, **GetDataValueEx method**, **MoveToField method**

SetDataValueEx method

Description

Sets the value in the specified currency field. The data value supplied must be in the form indicated by the format selector.

Syntax

object.**SetDataValueEx(***field_name*, *string_value*, *format_selector***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *field_name* – A string containing the name of the currency field for which the data value will be set. The field name must be fully qualified.
- *string_value* – A string containing the value to which you want to set the currency field.
- *format_selector* – An integer indicating the format of the string containing the value for the field. The following table lists the integer values and the corresponding format.

Integer value	Format
0	Control Panel Defaults
1	1,234.
2	1,234.5
3	1,234.56
4	1,234.567
5	1,234.5678
6	1,234.56789
7	$1,234.
8	$1,234.5
9	$1,234.56
10	$1,234.567
11	$1,234.5678
12	$1,234.56789

Return value

An integer indicating whether the field value was set. The value 0 indicates the field value was set. Any other value indicates the field value was not set.

Examples

This example uses the **SetDataValueEx method** to set the value of the Sales Amount currency field in the Receivables Transaction Entry window. The value supplied has two decimal places and doesn't include the currency symbol.

```
Private Sub SetDataValueEx_Click()
    Dim ErrVal As Integer
    ErrVal = GPApp.SetDataValueEx( _
    "'Sales Amount' of window RM_Sales_Entry" & _
    "of form RM_Sales_Entry", "145.85", 3)
End Sub
```

Related items

Commands

GetDataValue method, GetDataValueEx method, SetDataValue method

SetParamHandler method

Description

Specifies the object in the integrating application that will be used as the parameter handler to exchange values between the integrating application and pass-through sanScript.

Syntax

object.**SetParamHandler(***paramhandler_object***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *paramhandler_object* – The name of the object in the integrating application that will be used to pass values between the integrating application and pass-through sanScript.

Return value

The integer value 0.

Comments

You must use this method to specify the parameter handler object before you can pass values between the integrating application and pass-through sanScript.

Examples

This example uses the **SetParamHandler method** to set the ParamHandler object as the parameter handler object.

```
Private Sub SetParamHandler()
    Dim ErrVal As Integer
    ErrVal = GPApp.SetParamHandler(ParamHandler)
End Sub
```

Related items

Additional information
Chapter 4, "Passing Parameters"

Show method

Description

Causes Microsoft Dynamics GP to become visible if it was hidden.

Syntax

object.**Show**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Return value

None

Examples

The following example uses the **Show method** to make the Microsoft Dynamics GP application visible.

```
Public Sub ShowDynamicsGP_Click()
    GPApp.Show
End Sub
```

Related items

Commands

Hide method, Visible property

StartWizardMode method

Description

Causes Microsoft Dynamics GP to switch to the specified Wizard Mode, allowing the name of a resource to be retrieved.

Syntax

object.**StartWizardMode(***mode*, *callback_object*, *callback_method***)**

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *mode* – An integer that specifies which type of resource the user will select in Wizard Mode. The following table lists the modes:

Value	Description
1	Field mode. The user must click on a field in the application to return its fully-qualified name.
2	Window mode. The user must click on a window in the application to return its fully-qualified name.
3	Form mode. The user must click on a window in the application to return the name of the form the window is part of.

- *callback_object* – The name of the callback object in the integrating application containing the method to be run in response to the user clicking on an item while in Wizard Mode.
- *callback_method* – A string containing the name of the method that will be run in the callback object.

Return value

The integer value 0.

Comments

The method in the callback object must have two string parameters. The first parameter will be set to the fully-qualified resource name for the item the user clicked on while in Wizard Mode. The second parameter will be set to the name of the product containing the item clicked on.

Examples

This example uses the **StartWizardMode method** to look up the name of a field. The WizardCallback method in the GPCallback object will be run when the user clicks on a field in Microsoft Dynamics GP.

```
Private Sub StartWizardMode_Click()
    Dim i As Integer
    i = GPApp.StartWizardMode(1, GPCallback, "WizardCallback")
End Sub
```

The following is the code for the WizardCallback method contained in the GPCallback object. Note that it has two string parameters to which the fully-qualified resource name and product will be returned.

```
Public Sub WizardCallback(ResName As String, Product As String)
    NameWizard.ResourceName.Text = ResName
    NameWizard.Product.Text = Product
    GPApp.StopWizardMode
    NameWizard.Show
End Sub
```

Related items

Commands

StopWizardMode method

StopProcessing method

Description Causes Microsoft Dynamics GP to stop processing the current sequence of scripts.

Syntax *object*.**StopProcessing**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Comments This method is used in the callback procedure for focus triggers. If the focus trigger runs before the Microsoft Dynamics GP code, you may want to prevent the Microsoft Dynamics GP code from running. You would use the **StopProcessing method** to do this.

Examples This example uses the **StopProcessing method** in the callback procedure that runs in response to clicking the Save button in the Customer Maintenance window. The callback procedure runs *before* the Microsoft Dynamics GP code for the push button. The callback procedure ascertains whether the Comment1 field in the Customer Maintenance window contains data. If it doesn't, an error message is displayed and the Microsoft Dynamics GP Save Button script is prevented from running. The following is the code for the callback procedure.

```
Public Sub cbPushSaveButton()
    Dim ErrVal As Integer
    Dim ErrMsg As String
    If GPApp.GetDataValue("Comment1 of window " & _
    "RM_Customer_Maintenance of form RM_Customer_Maintenance") _
    = "" Then
        ErrVal = GPApp.ExecuteSanScript( _
        "error ""You must enter a comment."";", ErrMsg)
        GPApp.StopProcessing
    End If
End Sub
```

Related items

Commands
RegisterFocusTrigger method

StopWizardMode method

Description Causes Microsoft Dynamics GP to leave Wizard Mode.

Syntax *object*.**StopWizardMode**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Comments You must use this method after you have used the **StartWizardMode method**.

Examples The following is the code for the WizardCallback method contained in the GPCallback object. This method is run in response to the user clicking on a resource while in Wizard Mode. The **StopWizardMode method** is used after the name and location of the resource have been retrieved.

```
Public Sub WizardCallback(ResName As String, Product As String)
    NameWizard.ResourceName.Text = ResName
    NameWIzard.Product.Text = Product
    GPApp.StopWizardMode
    NameWizard.Show
End Sub
```

Related items

Commands
StartWizardMode method

VBE method

Description

Retrieves a reference to the Visual Basic for Applications (VBA) environment embedded in Microsoft Dynamics GP. The VBA environment must be open and active in Microsoft Dynamics GP for this method to work properly.

Syntax

object.**VBE**

Parameters

- *object* – A Microsoft Dynamics GP application object.

Return value

A reference to the Visual Basic for Applications environment.

Comments

To access the VBA environment, your integrating application should include the Microsoft Visual Basic for Applications Extensibility reference.

Examples

The following example uses the **VBE method** to retrieve a reference to the Visual Basic for Applications environment within Microsoft Dynamics GP. Then the version number of the VBA environment is retrieved and displayed to the user.

```
Public Sub VBE_Click()
    Dim VBE As Object
    Set VBE = GPApp.VBE
    MsgBox VBE.Version
End Sub
```

Visible property

Description

Returns a boolean indicating whether the application is visible, or makes the application visible or invisible.

Syntax

object.**Visible** [=*status*]

Parameters

- *object* – A Microsoft Dynamics GP application object.
- *status* – A boolean indicating whether the application should be visible or invisible. True indicates the application will be visible, while false indicates it will not.

Return value

Boolean

Examples

The following example uses the **Visible property** to find out whether the Microsoft Dynamics GP application is visible. If it is not, a message is displayed.

```
Public Sub IsDynamicsGPVisible_Click()
    If GPApp.Visible = False Then
        MsgBox "Microsoft Dynamics GP is hidden."
    End If
End Sub
```

The following example uses the **Visible property** to make Microsoft Dynamics GP visible.

```
Public Sub MakeVisible_Click()
    GPApp.Visible = True
End Sub
```

Chapter 12: Field Object

Your integrating application will use the field object to ascertain characteristics of fields in Microsoft Dynamics GP. The properties that apply to the Field object are listed below. A detailed explanation of each appears on the following pages:

- Height property
- Left property
- MaxLength property
- PromptHeight property
- PromptLeft property
- PromptName property
- PromptStyle property
- PromptTop property
- PromptType property
- PromptWidth property
- Style property
- SubType property
- Top property
- Type property
- Width property

Height property

Description Returns an integer containing the height of the field, measured in pixels.

Syntax *object*.**Height**

Parameters

- *object* – A field object.

Examples This example uses the **Height property** to retrieve the height of the Customer Number field.

```
Private Sub Height_Click()
    Dim FieldObj As Object
    Dim FieldHeight As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    FieldHeight = FieldObj.Height
End Sub
```

Left property

Description

Returns an integer containing the position of the left edge of the field, measured in pixels from the left edge of the window.

Syntax

object.**Left**

Parameters

- *object* – A field object.

Examples

This example uses the **Left property** to retrieve the position of the left edge of the Customer Number field.

```
Private Sub Left_Click()
    Dim FieldObj As Object
    Dim FieldLeftPos As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    FieldLeftPos = FieldObj.Left
End Sub
```

MaxLength property

Description

Returns an integer containing the keyable length of the field. The keyable length is the number of characters that can be typed in the field.

Syntax

object.**MaxLength**

Parameters

- *object* – A field object.

Examples

This example uses the **MaxLength property** to retrieve the keyable length of the Customer Number field.

```
Private Sub MaxLength_Click()
    Dim FieldObj As Object
    Dim KeyableLength As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    KeyableLength = FieldObj.MaxLength
End Sub
```

PromptHeight property

Description

Returns an integer containing the height of the prompt linked to the field, measured in pixels.

Syntax

object.**PromptHeight**

Parameters

- *object* – A field object.

Examples

This example uses the **PromptHeight property** to retrieve the height of the prompt linked to the Customer Number field.

```
Private Sub PromptHeight_Click()
    Dim FieldObj As Object
    Dim PromptHeight As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptHeight = FieldObj.PromptHeight
End Sub
```

PromptLeft property

Description

Returns an integer containing the position of the left edge of the prompt linked to the field, measured in pixels from the left edge of the window.

Syntax

object.**PromptLeft**

Parameters

- *object* – A field object.

Examples

This example uses the **PromptLeft property** to retrieve the position of the left edge of the prompt for the Customer Number field.

```
Private Sub PromptLeft_Click()
    Dim FieldObj As Object
    Dim PromptLeftPos As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptLeftPos = FieldObj.PromptLeft
End Sub
```

PromptName property

Description

Returns a string containing the text of the prompt linked to the field.

Syntax

object.**PromptName**

Parameters

- *object* – A field object.

Comments

For check boxes and push buttons, the text appearing on the control is returned. For other fields that don't have prompts linked to them, the empty string "" is returned.

Examples

This example uses the **PromptName property** to retrieve the text for the prompt linked to the Customer Number field.

```
Private Sub PromptName_Click()
    Dim FieldObj As Object
    Dim PromptName As String
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptName = FieldObj.PromptName
End Sub
```

PromptStyle property

Description

Returns a long integer indicating the characteristics of the prompt linked to the field.

Syntax

object.**PromptStyle**

Parameters

- *object* – A field object.

Comments

This property provides information that is specific to the internal implementation of fields for the Microsoft Dynamics GP runtime engine. The individual characteristics are retrieved from the **PromptStyle property** through the use of *bitmasks*. A bitmask allows you to ascertain whether specific bits of the 32-bit style value are set. Each bitmask corresponds to a specific set of characteristics for the prompt.

To use a bitmask, you perform a logical AND operation on the style value and the bitmask value. If the operation results in a non-zero value, the characteristics that correspond to the bitmask have been applied to the prompt.

Most of the information contained in the return value is not useful for integrating applications. One useful characteristic that can be retrieved from the **PromptStyle property** is whether a field is required to contain data before Microsoft Dynamics GP will allow the window's contents to be saved. The example for this property demonstrates how to use a bitmask to retrieve whether a field is a required field.

Examples

The following example uses the **PromptStyle property** to ascertain whether the Customer Number field is required. The bitmask for the **PromptStyle property** indicating that a field is required is the hexadecimal value 200. (It is common for bitmasks to be represented in hexadecimal form.) An AND operation is performed on the prompt style value and the bitmask. If a non-zero value is the result, the field is required.

```
Public Sub IsRequired_Click()
    Dim FieldObj As Object
    Dim FieldRequired As Boolean
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    'Is the field required?
    If (FieldObj.PromptStyle And &H200) <> 0 Then
        FieldRequired = True
    Else
        FieldRequired = False
    End If
End Sub
```

PromptTop property

Description

Returns an integer containing the position of the top edge of the prompt linked to the field, measured in pixels from the top of the window.

Syntax

object.**PromptTop**

Parameters

- *object* – A field object.

Examples

This example uses the **PromptTop property** to retrieve the position of the top edge of the prompt for the Customer Number field.

```
Private Sub PromptTop_Click()
    Dim FieldObj As Object
    Dim PromptTopPos As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptTopPos = FieldObj.PromptTop
End Sub
```

PromptType property

Description

Returns an integer indicating the type of prompt linked to the field.

Syntax

object.**PromptType**

Parameters

- *object* – A field object.

Comments

The value 0 indicates no prompt is attached to the field. The value 12 indicates the prompt is a string. Currently, no other return values are possible.

Examples

This example uses the **PromptType property** to retrieve the type of prompt for the Customer Number field. If the value is not 0, the field has a prompt and the prompt name is retrieved.

```
Private Sub PromptType_Click()
    Dim FieldObj As Object
    Dim PromptType As Integer
    Dim PromptName As String
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptType = FieldObj.PromptType
    If PromptType <> 0 Then
        'A prompt is attached to the field
        PromptName = FieldObj.PromptName
    End If
End Sub
```

PromptWidth property

Description

Returns an integer containing the width of the prompt linked to the field, measured in pixels.

Syntax

object.**PromptWidth**

Parameters

- *object* – A field object.

Examples

This example uses the **PromptWidth property** to retrieve the width of the prompt linked to the Customer Number field.

```
Private Sub PromptWidth_Click()
    Dim FieldObj As Object
    Dim PromptWidth As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
     "of form RM_Customer_Maintenance")
    PromptWidth = FieldObj.PromptWidth
End Sub
```

Style property

Description Returns a long integer indicating the characteristics of the field.

Syntax *object*.**Style**

Syntax *object* – A field object.

Comments This property provides information that is specific to the internal implementation of fields for the Microsoft Dynamics GP runtime engine. The individual characteristics are retrieved from the **Style property** through the use of *bitmasks*. A bitmask allows you to ascertain whether specific bits of the 32-bit style value are set. Each bitmask corresponds to a specific set of characteristics for the field.

To use a bitmask, you perform a logical AND operation on the style value and the bitmask value. If the operation results in a non-zero value, the characteristics that correspond to the bitmask have been applied to the field.

Most of the information contained in the return value is not useful for integrating applications. Two useful characteristics that can be retrieved from the **Style property** are whether a field is editable by the user and whether a list field is sorted. The examples for this property demonstrate how to use bitmasks to retrieve these characteristics.

Examples Style properties for fields are often retrieved when a field is selected while in Wizard Mode. The following code starts Wizard Mode. The WizardCallback method in the GPCallback object will be run when the user clicks on a field in Microsoft Dynamics GP.

```
Private Sub SelectField_Click()
    Dim i As Integer
    i = GPApp.StartWizardMode(1, GPCallback, "WizardCallback")
End Sub
```

The following is the code for the WizardCallback method contained in the GPCallback object. It creates a field object based upon the name information returned from Wizard Mode. The **Style property** is used to retrieve the characteristics of the field the user clicked on. The bitmask indicating a disabled appearance is the hexadecimal value 1. (It is common for bitmasks to be represented in hexadecimal form.) An AND operation is performed on the style value and the bitmask. If a non-zero value is the result, the field is disabled and can’t be edited.

```
Public Sub WizardCallback(ResName As String, Product As String)
    Dim FieldObj As Object
    'Stop Wizard Mode
    GPApp.StopWizardMode
    'Create the field object
    Set FieldObj = GPApp.CreateFieldObject(ResName)

    'Is the field editable by the user?
    If (FieldObj.Style And &H1) <> 0 Then
        'Field is disabled. Can't set its value.
        MsgBox "Field is disabled. Can't set its value."
    End If
End Sub
```

The following example is the code for the WizardCallback method contained in the GPCallback object. It creates a field object based upon the name information returned from Wizard Mode. The **SubType property** is used to ascertain whether the selected field is a list field (list box, drop-down list, multi-select list, button drop list, combo box or visual switch). If the field is a list, the **Style property** is used to ascertain whether the items in a list field are sorted. The bitmask indicating whether items are sorted is the hexadecimal value 10000.

```
Public Sub WizardCallback(ResName As String, Product As String)
    Dim FieldObj As Object
    Dim SortedList As Boolean
    'Stop Wizard Mode
    GPApp.StopWizardMode
    'Create the field object
    Set FieldObj = GPApp.CreateFieldObject(ResName)

    'Is the field a list?
    Select Case FieldObj.SubType
        Case 6, 7, 8, 13, 14, 21, 25
        If (FieldObj.Style And &H10000) <> 0 Then
            SortedList = True
        Else
            SortedList = False
        End If
    End Select
End Sub
```

SubType property

Description Returns an integer indicating the control type of the field.

Syntax *object*.**SubType**

Parameters

- *object* – A field object.

Comments The following table lists the values that are returned for the various field types:

Field type	**Value**
Button drop list	25
Check box	12
Combo box	13
Composite	15
Currency	2
Date	18
Drop-down list	6
Horizontal list box	17
Integer	0
List box	8 (non-native) or 21 (native)
Long integer	1
Multi-select list box	7
Picture	23
Progress indicator	22
Push button	16
Radio group	9
String	4
Text	5
Time	19
Visual switch	14

Examples This example uses the **SubType property** to retrieve the field type of the Customer Number field.

```
Private Sub SubType_Click()
    Dim FieldObj As Object
    Dim FieldType As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    FieldType = FieldObj.SubType
End Sub
```

Top property

Description

Returns an integer containing the position of the top edge of the field, measured in pixels from the top of the window.

Syntax

object.**Top**

Parameters

- *object* – A field object.

Examples

This example uses the **Top property** to retrieve the position of the top edge of the Customer Number field.

```
Private Sub PromptTop_Click()
    Dim FieldObj As Object
    Dim PromptTopPos As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    PromptTopPos = FieldObj.Top
End Sub
```

Type property

Description Returns an integer indicating the primary category of the field.

Syntax *object*.**Type**

Parameters

- *object* – A field object.

Comments The following table lists the values that are returned for the various types of fields:

Field type	Value
Button drop list	33
Check box	18
Combo box	34
Composite	36
Currency	21
Date	21
Drop-down list	30
Horizontal list box	24
Integer	21
List box	25 (non-native) or 32 (native)
Long integer	21
Multi-select list box	31
Picture	9
Progress indicator	20
Push button (with native picture)	17
Push button (with text)	29
Radio group	23
String	21
Text	22
Time	21
Visual switch	27

*The same value is returned for composites, currency values, dates, integers, long integers, strings and times. You can use the **SubType property** to differentiate between these fields.*

Examples This example uses the **Type property** to retrieve the primary category of the Customer Number field.

```
Private Sub Type_Click()
    Dim FieldObj As Object
    Dim FieldType As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    FieldType = FieldObj.Type
End Sub
```

Width property

Description Returns an integer containing the width of the field, measured in pixels.

Syntax *object*.**Width**

Parameters

- *object* - A field object.

Examples This example uses the **Width property** to retrieve the width of the Customer Number field.

```
Private Sub Width_Click ()
    Dim FieldObj As Object
    Dim FieldWidth As Integer
    Set FieldObj = GPApp.CreateFieldObject _
    ("'Customer Number' of window RM_Customer_Maintenance " & _
    "of form RM_Customer_Maintenance")
    FieldWidth = FieldObj.Width
End Sub
```

Glossary

Automation
An industry-standard technology that allows applications to provide access to the objects in the application. See also *object*.

Automation client
An application that accesses objects from an automation server. See also *Automation server*.

Automation server
An application that provides access to its objects through automation. See also *Automation client*.

Callback
The process of Microsoft Dynamics GP calling a method in a callback object when an event occurs in Microsoft Dynamics GP.

Callback class
The class in the project that contains the methods that run as a result of triggers from events in Microsoft Dynamics GP.

Callback object
The object in the integrating application that is created from the callback class.

Continuum Integration Library
The code in Microsoft Dynamics GP that describes the objects Microsoft Dynamics GP makes available to other applications through COM. Other applications can use these objects when they integrate with Microsoft Dynamics GP.

Database-level integration
An integration in which the integrating application reads from or writes to the Microsoft Dynamics GP database.

DEX.TLB
The type library that contains definitions of the objects, methods and properties available in Microsoft Dynamics GP. See also *TLB file*.

Dexterity
The application development tool used to create Microsoft Dynamics GP.

Interface-level integration
An integration where the integrating application keeps information synchronized with windows displayed in Microsoft Dynamics GP.

Object
A combination of code and data that contains information about an application or an item in the application, such as a control or window.

Parameter handler
A mechanism that allows the integrating application and pass-through sanScript to exchange data.

Parameter handler class
A class in the integrating application that defines the properties and methods that will be used to pass data between integrating application and pass-through sanScript.

Parameter handler object
The object created from the parameter handler class defined in the integrating application. Both the integrating application and pass-through sanScript have access to the properties and methods in this object.

Pass-through sanScript
SanScript code that is embedded in the code of the integrating application. The sanScript code is passed into the Microsoft Dynamics GP runtime engine, where it is compiled and executed.

Process-level integration
An integration in which the integrating application updates information whenever a Microsoft Dynamics GP process, such as posting, is performed.

TLB file
The file extension for a type library. A type library contains definitions of the objects, methods and properties an application exposes through COM.

Trigger
An event occurring in Microsoft Dynamics GP that causes methods in the integrating application to be run. See also *callback*.

Index