Microsoft Dynamics™ GP

Continuum sanScript Supplement

Publication date

February 2007

Contents

INTRODUCTION

Introduction

This supplement is intended for programmers who are not familiar with sanScript. It contains information about core sanScript functionality used to create database-level and process-level integrations with the Continuum API.

This supplement does not provide information about the Microsoft Dynamics GP database structure and processes. The Microsoft Dynamics GP Software Development Kit (SDK) contains that type of information. You will need the information in the Microsoft Dynamics GP SDK to effectively integrate with Microsoft Dynamics GP.

What's in this document

The manual is divided into the following parts:

- Part 1, **Scripting**, introduces the sanScript language and explains how to use sanScript in your Continuum integrations.
- Part 2, **sanScript Reference**, describes the commands in the sanScript language necessary to create database-level and process-level integrations. It also provides a list of data types and alert messages for sanScript.

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 when completing tasks with Integration Assistant for Excel.
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 6, "Commands"	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.
Software Development Kit (SDK)	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: SCRIPTING

Part 1: Scripting

This part introduces sanScript and describes how to use it to create database-level and process-level integrations with Continuum. The information is divided into the following chapters:

- Chapter 1, "sanScript," introduces the sanScript language.
- Chapter 2, "Working With Data," describes how to use the various data types in sanScript.
- Chapter 3, "Database-level Integrations," provides information about how to perform database operations with sanScript.
- Chapter 4, "Process-level Integrations," explains how to use Microsoft Dynamics GP procedures and functions for process-level integrations with sanScript.

Chapter 1: sanScript

SanScript is the language used by the Dexterity® development system. It is a powerful, English-like language that is designed to be easy to learn. The code for all Dexterity-based applications like Microsoft Dynamics™ GP, is written in sanScript.

With the Continuum API, you have the ability to write a sanScript script and pass it into Microsoft Dynamics GP where it will be compiled an executed. This pass-through sanScript allows you to access the power of sanScript from within your Continuum application.

This chapter contains information about the following:

General syntax

SanScript is a relatively easy language to learn, but you must follow some basic rules when you write scripts.

- SanScript is case-sensitive, as are names of fields, tables and other resources. All keywords in sanScript must be lowercase.
- Each statement in a script must end with a semi-colon. Statements can span multiple lines.
- Comments can appear anywhere in a script. They must be bounded by braces – { } – and can span multiple lines.
- If a script has local variables, these must be declared at the beginning of the script before any other lines in the script.

Names

In a script, you will refer to the names of items such as fields and tables in the Microsoft Dynamics GP application dictionary. The names of these items must be fully qualified so the items can be properly located.

A qualified name consists of a *qualifier*, which specifies the location of the item, and the item's name. For example, to refer to the Customer Name field in the RM_Customer_MSTR table, you would qualify the field with the name of the table where the field is located. The fully-qualified name is shown in the following script.

```
set 'Customer Name' of table RM_Customer_MSTR to "Bob Smith";
```

When you reference an item whose name contains spaces, you must enclose the name inside single quotation marks. The "of" portion of the name shouldn't be inside the quotation marks. The following name is valid:

This name	Is valid because
'Customer Name' of table RM_Customer_MSTR	The "of" portion of the name is not within the single quotation marks.

The following name isn't valid:

This name	Isn't valid because
'Customer Name of table RM_Customer_MSTR'	The "of" portion of the name is within the single quotation marks.

Single and double quotation marks are used differently in scripts. Single quotation marks indicate the names of objects, while double quotation marks indicate that the item between the quotation marks is a string value.

Tables

When you refer to a table, you must use the *table* qualifier. For example, the following sanScript code counts the number of records in the RM_Customer_MSTR table.

```
set record_count to countrecords(table RM_Customer_MSTR);
```

Table fields

When you refer to a field in a table, you must specify the table in which the field is located. For example, the following sanScript code refers to the Document Number field in the RM_Sales_WORK table.

```
set doc_val to 'Document Number' of table RM_Sales_WORK;
```

Statements

Statements are a type of command used in sanScript to complete a specific action in your application, such as saving an item in a table. For instance, you will use the **save table** statement to save a record in a table:

```
save table RM_Customer_MSTR;
```

Functions

Functions are commands similar to statements, but unlike statements, they return a value that is then used by another portion of the script. For example, the **countrecords()** function counts the number of records in a table. The name of the table is the function's parameter while the number of records in the table is the value returned. The following example sets the variable "number" to the number of records in the RM_Customer_MSTR table.

```
set number to countrecords(table RM_Customer_MSTR);
```

Data types

Dexterity uses several data types to specify how information is stored. When writing scripts, it's important to know how information in a field using a particular data type is stored, so you will know how script commands will control the information.

Fields store data in the following standard forms: boolean, currency, integer, long, string, text, date and time. The following table lists each storage type, its description, and the data types that use that storage type.

Storage type	Description	Data types using this storage type
Boolean	A value of either true or false.	Boolean Check box
Currency	A currency value in the range [-99,999,999,999,999.99999 to 99,999,999,999,999.99999]. The decimal point is implied in the number, but not actually stored. For display purposes, currency values are limited to 14 digits to the left of the decimal and 5 digits to the right.	Currency
Integer	An integral number in the range [-32767 to 32767].	Drop-down list Group box Integer List box Visual switch
Long	An integral number in the range [-2,147,483,648 to 2,147,483,647].	Long integer
String	A sequence of up to 255 characters.	Combo box String
Text	A sequence of up to 32,000 characters.	Text
Date	Date based on the Julian calendar.	Date
Time	Time based on the 24-hour standard.	Time

The other data types not mentioned in the previous table store information in a format unique to the data type. Composite and multi-select list box data types each store data in their own unique format.

Variables

Variables allow an application to temporarily store values used by the application. SanScript has two types of variables: local variables and global variables.

Local variables

Local variables are specific to a single script and are active only while the script is running. They're used to store intermediate values, such as the resulting value of an expression, while a script is running. To create local variables, define them at the beginning of the script they'll be used in. The word **local**, followed by the type of variable (integer, long, currency, string, time, boolean) and the variable name are required for each local variable.

*Note that you use the word **long** to indicate a long integer type in scripts.*

In the following example, a local long integer variable named "number_of_records" is used to store a temporary value for the script. The **set** statement is used to set the value of the variable to the value returned by the **countrecords()** function.

```
local long number_of_records;

set number_of_records to countrecords(table RM_Customer_MSTR);
```

Global variables

Global variables are active the entire time the Microsoft Dynamics GP application is open, so they're available to any script at any time. Global variables are used to store information that affects the entire application. To reference global variables in a script, the qualifier **of globals** must appear after the name of the global variable.

In the following example, the global variable User ID is checked retrieved to see who is currently logged into Microsoft Dynamics GP.

```
local string user_ID;

set user_ID to 'User ID' of globals;
```

Constants

Constants are fixed numeric or string values that are used in scripts, where the constant name is used in place of the value associated with it. Two types of constants are available: predefined constants and user-defined constants.

Predefined constants

SanScript has a number of predefined constants that can be used in scripts. For example, the three constants listed in the following table are associated with the **ask()** function.

Constant name	Description
ASKBUTTON1	The value returned from the **ask()** function when the first button is clicked in the ask dialog box.
ASKBUTTON2	The value returned from the **ask()** function when the second button is clicked in the ask dialog box.
ASKBUTTON3	The value returned from the **ask()** function when the third button is clicked in the ask dialog box.

In the following example, the **ask()** function uses predefined constants to find out which button the user clicked.

```
if ask("Include sales tax?", "Yes", "No") = ASKBUTTON1 then
    set Total to Total + Tax;
end if;
```

User-defined constants

Microsoft Dynamics GP also has its own constants that were added when the application was coded. For example, the following script retrieves the constant corresponding to the number of periods.

```
local integer periods;

set periods to NUMBER_PERIODS;
```

Expressions

An expression is a sequence of *operands* and *operators* that are evaluated to return a value. Operators indicate the type of procedure to perform on the operands. The operands are usually items such as fields or the values returned from functions or other expressions. There are four kinds of expressions in sanScript: numeric, date and time, string, and boolean.

Numeric expressions

Evaluating a numeric expression results in numeric value. For instance, the following sanScript statement uses a numeric expression that returns the total price of a sale by adding the subtotal and sales_tax values.

```
set total to subtotal + sales_tax;
```

The following table lists the numeric operators in the order they're evaluated.

Operator	Use	Example
unary minus (-)	negation	-15
^	power	10^3
*, /,%	multiplication, division and modulus	15/3
+,-	addition and subtraction	5+10
=,<>,<,>,<=,>=	comparison	4 >= 2

Date and time expressions

Evaluating a date and time expression results in a date, time or a numeric value. For instance, the following sanScript statement adds 30 days to the value of the system date to set the due_date variable.

```
set due_date to sysdate() + 30;
```

The following table lists the date and time operators in the order they're evaluated.

Operator	Use	Example
+,-	addition and subtraction	sysdate() + 30
=,<>,<,>,<=,>=	comparison	sysdate() <= 'Due Date'

String expressions

Evaluating a string expression results in a string value. For instance, the following sanScript statement joins two string values and stores the result in the user_name variable.

```
set user_name to "Cindy " + "Johnson";
```

The following table lists the string operators in the order they're evaluated.

Operator	Use	Example
+	concatenation	"Sara " + "Johnson"
=,<>,<,>,<=,>=	comparison	'Name' <> "Smith"

Boolean expressions

Evaluating a boolean expression results in boolean value of true or false. Boolean expressions are used in decision-making statements such as the **if...then** statement.

For instance, the following sanScript statement evaluates the boolean expression in the **if...then** statement to find out whether the script should continue, based on who the current user is.

```
if 'User ID' of globals = "SKUBIS" then
    {Don't allow the script to proceed.}
    abort script;
end if;
```

The following table lists the boolean operators in the order they're evaluated.

Operator	Use	Example
not	logical not	not (err() = 0)
=,<>	equal, not equal	(6 + 4) = (5 + 5)
and	logical and	(Amount > 10) and (Amount < 20)
or	logical or	(Password = "Smith") or (Password = "smith")

Overflow in numeric expressions

Overflow in a numeric expression occurs when an intermediate or final result of the expression is too large to be stored by the type of data used in the expression. This is especially common with integer values.

For example, multiplying the integers 450 and 75 should result in the value 33,750, but instead results in -31,786. Because the actual result is larger than 32,767, the maximum amount that can be represented by an integer, overflow occurs.

One method of preventing overflow is to convert all integer values in the expression to long integers and then evaluate the expression. All integers in the expression, not just the final result, must be converted to long integers to avoid overflow. This is because the overflow can occur in intermediate steps while evaluating the expression, not only the final step. For example, the expression in the following script would still overflow:

```
local long product;
set product to 450 * 75;
```

To avoid the overflow, the two integer values must also be converted to long integers as shown in the following script:

```
local long product, operand1, operand2;
set operand1 to 450;
set operand2 to 75;
set product to operand1 * operand2;
```

Operators

The following table lists the operators supported in sanScript. Examples are included to show how each operator is used in the different types of expressions.

Operator	Description	Example
unary minus (-)	In numeric expressions, the unary minus operator (-) indicates a negative value.	set neg_val to –10;

Operator	Description	Example
addition (+)	In numeric expressions, the result is the sum of the two values.	set total to subtotal + 5;
	In string expressions, the result is the concatenation of the first and second strings.	set name to "Steve " + "Anderson";
	In date expressions, a numeric quantity may be added to a date to form a new date. The numeric value is treated as a number of days.	set due_date to sysdate() + 30;
	In time expression, a numeric quantity may be added to a time value to form a new time value. The numeric value is treated as a number of minutes.	set next_hour to systime() + 60;
subtraction (-)	In numeric expressions, the result is the difference of the two values.	set total to price - discount;
	In date expressions, a numeric quantity may be subtracted from a date value to form a new date value. The numeric value is treated as a number of days.	set yesterday to sysdate() – 1;
	Two date values can be subtracted to find the difference between them in days.	set days_to_pay to due_date - sysdate();
	In time expressions, a numeric quantity may be subtracted from a time value. The numeric value is treated as a number of minutes.	set start_time to systime() – 30;
subtraction (-)	Two time values can be subtracted to find the difference between them in minutes.	set elapsed_time to start_time – end_time;
multiplication (*)	The multiplication operator (*) is used in numeric expressions. The result is the product of the two numbers.	set total to number_of_periods * contribution;
division (/)	The division operator (/) is used in numeric expressions. The result is the quotient of the two numbers.	set check_total to salary / number_of_pay_periods;
modulus (%)	The modulus operator (%) is used in numeric expressions. The result is the remainder of the division of the first number by the second number (for example, 71 % 10 = 1).	set single_items to items_ordered % items_per_box;
power (^)	The power operator is used in numeric expressions. The result is the first operand raised to the power of the second operand. Only powers of 10 may be calculated.	set number to 10 ^ 5;

equality (=)	The equality operator is supported in numeric, boolean, string, date and time expressions. In all expressions the result is true if the two operands are equal, and false if they are not equal.	if total = 100 then
		  set result to true;
		end if;

inequality (<>)	The inequality operator is supported in numeric, boolean, string, date and time expressions. In all expressions the result is false if the two operands are equal, and true if they are not equal.	if password <> "access" then
		  abort script;
		end if;

less than (<)	The less than operator is supported in numeric, string, date and time expressions. In all expressions the result is true if the first operand is less than the second operand, and false if it is not.	if total < 100 then
		  warning "Total is not 100%";
		end if;

Operator	Description	Example

greater than (>)	The greater than operator is supported in numeric, string, date and time expressions. In all expressions the result is true if the first operand is greater than the second operand, and false if it is not.	if current_date>sysdate then
		  warning "Date is not valid.";
		end if;

less than or equal to (<=)	The less than or equal to operator is supported in numeric, string, date and time expressions. In all expressions the result is true if the first operand is less than or equal to the second operand, and false if it is not.	if systime() <= posting_time
		  then
		  warning "Posting can't begin.";
		end if;

greater than or equal to (>=)	The greater than or equal to operator is supported in numeric, string, date and time expressions. In all expressions the result is true if the first operand is greater than or equal to the second operand, and false if it is not.	if total >= 100 then
		  set discount to 10;
		else
		  set discount to 0;
		end if;

and	The and operator is supported in boolean expressions. The result is true if both the operands are true, and false if either of the operands is false.	if (count> 100) or (total> 0) then
		  set discount to 20;
		end if;

or	The or operator is supported in boolean expressions. The result is true if either the operands is true, and false if both of the operands are false.	if (count> 100) or (total> 100)
		  then set discount to 20;
		end if;

not	The not operator is used in boolean expressions to complement (reverse) the value of a boolean expression.	if not taxable, then
		  set total to subtotal;
		else
		  set total to subtotal + tax;
		end if;

Arrays

The individual pieces that make up an array are called *elements*. The elements are numbered from 1 to the size of the array. Arrays can have up to 32,767 elements. The number used to indicate a specific element of the array is called the *array index*. A specific element of the array is referenced in sanScript using its corresponding array index.

Dexterity-based applications can use array fields and array local variables. Array fields must have been created when the Dexterity-based application was coded. Array local variables are created by including the size of the array in brackets – [ ] – following the local variable name.

To access the elements of an array from within a script, simply use the name of the array and add the index number of the element you want to reference. Be sure the index number is included in square brackets after the name of the array and before the qualifier. The following example sets the second element of the History Figures array to 5500.

```
set 'History Figures'[2] of table GL_Account_SUM_HIST to 5500;
```

The following example uses a local array field with five elements to act as status flags for the script. The **for** loop initializes the flags to false.

```
local boolean status[5];
local integer i;

for i = 1 to 5 do
    set status[i] to false;
end for;
```

Composites

A composite is a special field that is actually composed of several individual fields. Each part of the composite is called a component. Composites can be used to store information such as account or item numbers, which are composed of several parts.

In sanScript, you can reference a composite field as a single entity. You can also reference the value of a component within the composite field. Referencing components of a composite is similar to referencing the elements of an array. The components of a composite are numbered from 1 to the total number of components in the composite. A specific component is referenced using the **component** keyword followed by the number of the component enclosed in parentheses.

For example, the following script sets the local variable segment_2 to the value of the second component of the Account Number composite field.

```
local string segment_2;

set segment_2 to str(component(2) of field 'Account Number' of table
GL_Account_MSTR);
```

Chapter 2: Working With Data

Whether you're creating a database-level or process-level integration, you will need to work with data in sanScript. This chapter describes how to convert data from one type to another. It also describes data type conversions necessary when you pass parameters to and from pass-through sanScript. Information is divided into the following sections:

Data type conversions

When you work with fields and variables in scripts, you may need to convert data from one storage type to another. This conversion can happen implicitly or explicitly.

Implicit conversions

Type conversion happens implicitly in many situations. For instance, if an integer value and a currency value are added together in an expression and the result is stored in a currency field, the result would be a currency. If the result of the same expression was stored in an integer field, the result would be an integer. The conversions that occur depend on where the result is stored.

Explicit conversions

To explicitly convert a string to a numeric value, use the **value()** function. To explicitly convert a numeric value to a string use the **str()** function. For more information about each of these functions, refer to Chapter 5, "Functions and Statements."

Passing parameters

The parameter handler object allows you to pass strings between your Continuum application and pass-through sanScript. If you want to pass other types of data, such as integers or date values, you will need to perform explicit data type conversions.

Passing data to sanScript

If you want to pass data other than strings to pass-through sanScript, you must first convert the data to string format in your Continuum application. Then you can pass the data to sanScript and reconvert it to the proper data type.

For example, assume you wanted to pass a date value from your Continuum application into pass-through sanScript. To do this, your Continuum application could divide the date into three strings, one containing the day, another the month and a third the year. You would use the parameter handler object to pass these three strings into pass-through sanScript. In sanScript, you would retrieve the three string parameters, convert them to numeric values using the **value()** function, and create a date value with the **setdate()** function.

Returning data from sanScript

Similar to passing data to sanScript, you can return only string values from pass-through sanScript. If you want to return other types of data, you must first convert the values to strings in sanScript, pass them back to your Continuum application, and then reconvert them to the proper type.

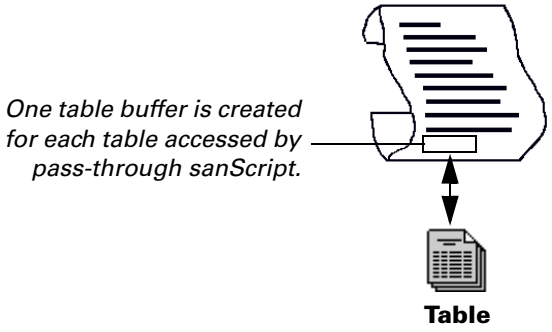
For example, if you wanted to return a time value from pass-through sanScript, you could use the **hour()**, **minute()**, and **second()** functions to retrieve the components of the time value. You would then use the **str()** function to convert the values to strings and then use the parameter handler object to pass them to your Continuum application. Your Continuum application would them have to convert them back into a time value.

Chapter 3: Database-level Integrations

When you create database-level integrations, you must work directly with Microsoft Dynamics GP tables. This chapter contains information you need to effectively work with tables in pass-through sanScript. The following topics are discussed:

Table buffers

A buffer is a temporary storage area in your computer's memory. Each time you access a table from pass-through sanScript, a *table buffer* is created for the table. A table buffer can hold one record of information. The information in the table buffer comes from either the table or the script, depending on whether you're reading a record from the table or adding new information to the table.



Tables are automatically opened when they are referenced in sanScript. They are closed when the pass-through sanScript has finished.

Common table operations

There are four common operations you will perform when working with tables:

- Retrieving a record
- Saving a new record
- Updating a record
- Removing a record

Retrieving a record

Retrieving a record is a multi-step process. You must first decide which key you want to use to retrieve the record. Use the Table Descriptions window in Microsoft Dynamics GP to ascertain the keys and key segments for each table. Then set the key field or fields and use the **get** or **change** statement to retrieve the record.

The following sanScript code retrieves the customer record for American Electrical. The second key of the RM_Customer_MSTR table is composed of the Customer Name field, so that key will be used to retrieve the record.

```
{Release the lock on any record currently in the table buffer.}
release table RM_Customer_MSTR;

{Step 1: Set the key fields that will be used to retrieve the record.}
set 'Customer Name' of table RM_Customer_MSTR to "American Electrical";

{Step 2: Use the second key to retrieve the record. Don't lock the record.}
get table RM_Customer_MSTR by number 2;

{Step 3: Check for any error that may have occurred.}
if err() = MISSING then
    error "The record was not found.";
end if;
```

Saving a new record

To save a new record, set the table fields to the values you want to save. Then use the **save table** statement to save the new record.

The following sanScript code saves a new entry in the GL_Account_Category_MSTR table. The **set** statement sets the new values in the table buffer, and then the **save table** statement saves the new record.

```
{Step 1: Set the values of the fields in the table buffer.}
set 'Account Category Number' of table GL_Account_Category_MSTR to 49;
set 'Account Category Description' of table GL_Account_Category_MSTR to
➥ "Profit Sharing";

{Step 2: Save the new record.}
save table GL_Account_Category_MSTR;

{Step 3: Check for any errors that may have occurred.}
if err() <> OKAY then
    error "An error occurred saving the new record: " + str(err());
end if;
```

Updating a record

To update an existing record, first read the record with the **change** statement. This retrieves the record and locks it, allowing you to make changes. Then use the **set** statement to make the changes to the appropriate fields in the record. Finally, use the **save table** statement to save the changed record back to the table.

The following sanScript code reads the customer record for Adam Park Resort. The Salesperson ID field is changed to STEVE K. Then the record is saved in the RM_Customer_MSTR table.

```
{Release the lock on any record currently in the table buffer.}
release table RM_Customer_MSTR;

{Step 1: Set the key fields that will be used to retrieve the record.}
set 'Customer Number' of table RM_Customer_MSTR to "ADAMPARK0001";

{Step 2: Use the first key to retrieve and lock the record, allowing it to be
changed.}
change table RM_Customer_MSTR by number 1;

{Step 3: Change the Salesperson ID.}
set 'Salesperson ID' of table RM_Customer_MSTR to "STEVE K.";

{Step 4: Save the changed record.}
save table RM_Customer_MSTR;

{Step 5: Check for any errors that may have occurred.}
if err() <> OKAY then
    error "An error occurred saving the record: " + str(err());
end if;
```

Removing a record

To remove a record from a table, first read the record with the **change** statement. This retrieves the record and locks it, allowing you to remove it with the **remove** statement.

The following sanScript code reads the first record in the RM_Customer_MSTR table and locks it. Then the **remove** statement removes the record from the table.

```
{Release the lock on any record currently in the table buffer.}
release table RM_Customer_MSTR;

{Step 1: Read and lock the first record in the table.}
change first table RM_Customer_MSTR;

{Step 2: Remove the record.}
remove table RM_Customer_MSTR;
```

Multiuser processing

Microsoft Dynamics GP supports multiple users accessing the same table at the same time. This is accomplished through Optimistic Concurrency Control (OCC), a form of record locking that allows multiple users to work in the same tables and access the same records with minimal restrictions, while helping to ensure data integrity.

Record locking

A record must be locked to delete it or save any changes made to it. A lock is applied when a record is read from a table. Two types of locks can be used: passive and active.

Passive locking

A *passive lock* allows other users to access the record. They can delete the record or make changes to it. Passive locking ensures that other users accessing the record can be made aware that the record has been deleted or that the contents of the record have changed. A passive lock is applied each time a record is read using the **change** statement.

Active locking

An *active lock* allows other users to read the record, but not make any changes or delete the record. Active locking ensures that the user who has the active lock is the only user who can make changes or delete the record. If other users try to delete or change the record, a table-sharing error will occur. An active lock is applied each time a record is read using the **change** statement with the **lock** keyword included.

*The **get** statement is used only to read a record. It never locks a record.*

Not all tables in Microsoft Dynamics GP allow active locking. If your pass-through sanScript code uses the **change** statement with the **lock** keyword on a table that doesn't allow active locking, you will receive a type incompatibility message.

Releasing locks

Any of the following actions releases a record lock:

- Using the **release table** statement.
- Using the **save table** or **remove** statements, regardless of whether the statement is successful.

If a record is currently locked in a table buffer, and you attempt to lock another record, you will receive an error message indicating that a record was already locked.

Writing your application

To allow multiple users to successfully use Microsoft Dynamics GP when pass-through sanScript code is running, you must choose the type of locking used as well as handle any error conditions that occur as a result of multiple users working with the same record.

The following table lists the various scenarios that can occur. The events listed happen in order from left to right. For example, in the first row User A passively locks a record, then User B passively locks the same record. User A deletes the record, then User B changes the contents of the record and saves the record. The changes User B made will be saved.

	User A	User B	User A	User B	Result
1	Passively locks a record.	Passively locks the same record.	Deletes the record.	Changes the contents of the record and saves the changes.	The changes User B made will be saved.
2	Passively locks a record.	Passively locks the same record.	Changes the contents of the record and saves the record.	Deletes the record.	The record will be deleted.
3	Passively locks a record.	Passively locks the same record.	Changes a field and saves the record.	Changes a different field and saves the record.	Both changes will be saved.
4	Passively locks a record.	Passively locks the same record.	Changes a field and saves the record.	Changes the same field and attempts to save the record.	User B will get an error indicating the record changed. User B's changes won't be saved.
5	Passively locks a record.	Passively locks the same record.	Deletes the record.	Attempts to delete the record.	User B will get an error indicating the record is missing.
6	Actively locks a record.	Passively locks the same record.	Keeps the active lock.	Attempts to delete the record or change a field and save the record.	User B will get a record locked error. The record won't be deleted or the changes won't be saved.
7	Actively locks a record.	Passively locks the same record.	Deletes the record. The active lock is released.	Changes the record and saves it or deletes the record.	If user B changed the record and saved, the changes will be saved. If User B attempts to delete the record, User B will get an error indicating the record is missing.
8	Actively locks a record.	Passively locks the same record.	Makes changes and saves the record. The active lock is released.	Changes the record and saves it or deletes the record.	If User B changed the same field as User A, User B will get an error indicating the record changed. User B's changes won't be saved. If user B changed different fields, the changes will be saved. If User B deleted the record, the record will be deleted.
9	Passively locks a record.	Actively locks the same record.	Attempts to delete the record or change a field and save the record.	Keeps the active lock.	User A will get a record locked error, even though User B's active lock came later than User A's lock.
10	Actively locks a record.	Attempts to actively lock the same record.			User B will get a record locked error.

Scenarios 1 through 3 don't produce any errors. To be multiuser compatible, your application should be able to handle scenarios 4 to 10, alerting users that an error occurred, and allowing them to respond appropriately.

The **err()** function is used to trap errors so the script can deal with the errors that occur. The following example scripts show how to trap for multiuser errors when reading, saving and removing records. They use the **err()** function to trap and handle errors.

You should not check for multiuser error conditions on tables containing text fields. Multiuser error codes are not properly returned for tables containing text fields.

*When you're writing scripts that handle errors using the **err()** function, you may want to use the **check error** statement as a debugging tool while you're working. The **check error** statement will display a message indicating the type of table error that occurred.*

Example 1

The following script reads and actively locks the first record in the RM_Sales_WORK table. It uses the **err()** function to handle an error resulting from the record being actively locked by another user.

```
{Release the lock on any record currently in the table buffer.}
release table 'RM_Sales_WORK';

{Read the first record in the table and actively lock it.}
change first table RM_Sales_WORK, lock;
if err() = LOCKED then
    {The record is actively locked by another user.}
    warning "This record is currently locked by another user.";
end if;
```

Example 2

The following script reads the first record in the RM_Customer_MSTR table, changes the Salesperson ID, and attempts to save the changed record. The **err()** function is used to handle an error resulting from the record being changed or being actively locked by another user.

```
{Release the lock on any record currently in the table buffer.}
release table 'RM_Sales_WORK';

{Read the first record in the table.}
change first table RM_Customer_MSTR;

{Change the Salesperson ID field.}
set 'Salesperson ID' of table RM_Customer_MSTR to "STEVE K.";

{Save the changed record.}
save table RM_Customer_MSTR;

if err() = RECORDCHANGED then
    {The record was changed by another user.}
    warning "This record has been changed by another user.
    ➥ Their change will be overwritten.";
    {Reread the current record to lock it.}
    change table RM_Customer_MSTR;
    set 'Salesperson ID' of table RM_Customer_MSTR to "STEVE K.";
    save table RM_Customer_MSTR;
    if err() <> OKAY then
        error "Customer record could not be updated.";
    end if;
elseif err() = LOCKED then
    {The record is actively locked by another user.}
    error "This record is actively locked by another user.
    ➥ Changes will not be saved.";
elseif err() <> OKAY then
    {Another table error occurred.}
    error "An error occurred saving the customer record: " + str(err());
end if;
```

Example 3

The following script reads the last record in the GL_Account_Category_MSTR table and attempts to delete it. The **err()** function is used to handle an error resulting from the record being actively locked or deleted by another user.

```
{Release the lock on any record currently in the table buffer.}
release table 'GL_Account_Category_MSTR';

{Read the last record in the table.}
change last table GL_Account_Category_MSTR;

if err() = OKAY then
    {The record was read and now can be removed.}
    remove table GL_Account_Category_MSTR;
    if err() = LOCKED then
        {The record was actively locked by another user.}
        error "The record is actively locked by another user
        ➥ and can't be deleted.";
    elseif err() = MISSING then
        {The record was deleted by another user.}
        warning "The record was already deleted by another user.";
    elseif err() <> OKAY then
        {Another table error occurred.}
        error "An error occurred deleting the record: " + str(err());
    end if;
elseif err() = LOCKED then
    {The record is actively locked by another user.}
    error "The account category record is locked by another user.";
elseif err() <> OKAY then
    {Another table error occurred.}
    error "An error occurred retrieving the record: " + str(err());
end if;
```

Ranges

When working with tables, it is often efficient to limit the amount of information being accessed. You can do this by setting up a *range* for the table. A range is based on a key for the table, and allows you to access a specified portion of the table. The selected range will be treated as an entire table. For instance, a **get first** statement returns the first record in the range, a **get last** statement returns the last record in the range, and so on.

You use the **range** statement to create a range for a table. You can specify one range per table, and the range is associated with a specific key. The range will be used only when the table is accessed by the key the range is associated with.

Example 1

In the following example, the **range** statement is used to limit the records accessed to only those customers whose names start with "A". Notice that the second key for the table, composed of the Customer Name field, is used for each of the **range** statements as well as the **get first** statement.

```
{Clear any existing range for the table.}
range clear table RM_Customer_MSTR;

{Set the start of the range.}
set 'Customer Name' of table RM_Customer_MSTR to "A";
range start table RM_Customer_MSTR by number 2;

{Set the end of the range.}
set 'Customer Name' of table RM_Customer_MSTR to "B";
range end table RM_Customer_MSTR by number 2;

{Read the first record in the range.}
get first table RM_Customer_MSTR by number 2;
```

Example 2

If a key is composed of several segments, you can create ranges based on several key segments. The **clear field** and **fill** statements are often used when setting ranges for multisegment keys. For example, the Purchase_Data table is shown in the following illustration. It has a key composed of the Purchase Date and the Store ID.

Purchase Date	Store ID	Amount
11/16/98	C	100.00
11/17/98	A	50.00
11/17/98	B	75.00
11/17/98	C	22.00
11/18/98	A	175.00
11/18/98	C	60.00
11/19/98	A	45.00
11/19/98	C	16.00
11/20/98	B	100.00

The following script sets a range to include all purchases made on 11/17/98 for all stores. The first segment of the key is set to the date 11/17/98. The second segment is set to its minimum value using the **clear field** statement, then the start of the range is set. The first segment remains 11/17/98. The second segment is set to its maximum value using the **fill** statement. Then the end of the range is set. Using the **clear field** and **fill** statements on the Store ID fields allows all stores to be selected.

```
{Clear any previous range for the table.}
range clear table Purchase_Data;

{Set the start of the range.}
set 'Purchase Date' of table Purchase_Data to
➥ setdate('Purchase Date' of table Purchase_Data, 11, 17, 1998);
clear field 'Store ID' of table Purchase_Data;
range start table Purchase_Data by number 1;

{Set the end of the range.}
set 'Purchase Date' of table Purchase_Data to
➥ setdate('Purchase Date' of table Purchase_Data, 11, 17, 1998);
fill 'Store ID' of table Purchase_Data;
range end table Purchase_Data by number 1;
```

Example 3

The following example deletes all customer information for Aaron Fitz Electrical. A single record must be deleted from the RM_Customer_MSTR and RM_Customer_MSTR_SUM tables. Several records must be deleted from the RM_Customer_MSTR_ADDR table. A range is used to delete these records.

```
{Delete the record from the RM_Customer_MSTR table.}
set 'Customer Number' of table RM_Customer_MSTR to "AARONFIT0001";
change table RM_Customer_MSTR by number 1;
if err() = OKAY then
    remove table RM_Customer_MSTR;
end if;

{Delete the record from the RM_Customer_MSTR_SUM table.}
set 'Customer Number' of table RM_Customer_MSTR_SUM to "AARONFIT0001";
change table RM_Customer_MSTR_SUM by number 1;
if err() = OKAY then
    remove table RM_Customer_MSTR_SUM;
end if;

{Delete the records from the RM_Customer_MSTR_ADDR table.}
{Clear any current range for the table.}
range clear table RM_Customer_MSTR_ADDR;

{Set the beginning of the range. The key has two segments. They are the
Customer Number and the Address Code.}
set 'Customer Number' of table RM_Customer_MSTR_ADDR to "AARONFIT0001";
clear field 'Address Code' of table RM_Customer_MSTR_ADDR;
range start table RM_Customer_MSTR_ADDR by number 1;

{Set the beginning of the range.}
set 'Customer Number' of table RM_Customer_MSTR_ADDR to "AARONFIT0001";
fill 'Address Code' of table RM_Customer_MSTR_ADDR;
range end table RM_Customer_MSTR_ADDR by number 1;

{Remove the range of records.}
remove range table RM_Customer_MSTR_ADDR;
```

Range types

How a range is evaluated depends on the database manager used for the table. For the Btrieve and c-tree database managers, ranges are evaluated *inclusively.* For the SQL database manager, ranges are evaluated either inclusively or *exclusively.*

Example 4

The following example illustrates the difference between these two methods of evaluating a range. The sample table shown in the following illustration has a key composed of the three segments shown.

Segment 1	Segment 2	Segment 3
A	A	A
A	A	B
A	A	C
A	B	A
A	B	B
A	B	C
A	C	A
A	C	B
A	C	C
B	A	A
B	A	B
B	A	C
B	B	A
B	B	B
B	B	C
B	C	A
B	C	B
B	C	C
C	A	A
C	A	B
C	A	C
C	B	A
C	B	B
C	B	C
C	C	A
C	C	B
C	C	C

The following range is set for the table:

Range start:	A A A
Range end:	A C B

The following records are included in the inclusive range.

Segment 1	Segment 2	Segment 3
A	A	A
A	A	B
A	A	C
A	B	A
A	B	B
A	B	C
A	C	A
A	C	B

The following records are included in the exclusive range.

Segment 1	Segment 2	Segment 3
A	A	A
A	A	B
A	B	A
A	B	B
A	C	A
A	C	B

Notice that inclusive and exclusive ranges don't contain the same records. This is an issue when developing an application, because the c-tree database manager always produce inclusive ranges, but the SQL database manager has optimal performance when it produces exclusive ranges. The SQL database manager can be forced to produce inclusive ranges, but the application's performance may be seriously degraded. To eliminate the discrepancies that result from the two types of ranges, we recommend that you create "well-behaved" ranges in your sanScript code.

"Well-behaved" ranges

A "well-behaved" range has the following characteristics:

1. Beginning with the leftmost key segment and working to the right, the first 0 to *n* segments are set to equal values for both the range start and the range end.

2. The next 0 or 1 segments are set to non-equal values for the range start and range end.

3. The remaining segments (if any) are cleared for the range start and filled for the range end.

The scripts in Examples 1, 2 and 3 all create "well-behaved" ranges. The ranges produced will be the same, regardless of the database manager being used.

Records with the same key values

Tables that contain records with the same key value must be handled cautiously. For example, in the table containing customer names, shown in the following illustration, the key is the customer's last name. Several customers have the last name "Smith". Thus, several records have the same key value.

Last Name1	First name
Schulz	Dan
Smith	Alan
Smith	Maria
Smith	Bob
Smith	Sharon
Thompson	Jean
Wallace	Phill

Adding records

To allow multiple records to have the same key value, the table must allow duplicates. Not all tables in Microsoft Dynamics GP allow duplicate records. If you add a duplicate record to a table that doesn't allow them, you will receive an error indicating a duplicate record was created. Use the **err()** function to trap for this error condition.

Retrieving records

It's more difficult to retrieve records that have the same key value from a table. The standard practice of setting the table buffer to the key value and then using a **get** or **change** statement won't work because the database manager can't guarantee which record will be retrieved. For example, if the table buffer for the table shown in the previous illustration was set to "Smith" and a **get** or **change** statement was used to retrieve a record, the table buffer could contain the record for Alan, Maria, Bob or Sharon Smith. There is no way of knowing which record will be read.

Two methods can be used to reliably retrieve records that have the same key value.

- Start at the beginning of the table using the **get first** or **change first** statement. Then use **get next** or **change next** statements to read all of the records in the table. Be sure you're using the *same* key for the **get first** or **change first** and **get next** or **change next** statements.

  This method will read all of the records in the table, including those that have the same key value. However, there is no way of determining the order in which the records having the same key value will be read.

- Use the **range start** statement to set the beginning of the range to the duplicate key value, and use the **range end** statement to set the end of the range to the duplicate key value. Use the **get first** or **change first** statement to retrieve the first record in the range. Then use the **get next** or **change next** statement to retrieve successive records from the range. Be sure you're using the *same* key for the **range** and **get** or **change** statements.

  This method will read all of the records that have the same key value specified using the **range** statements. However, there is still no way of determining the order in which the records having the same key value will be read.

The following script reads all of the "Smiths" from the example table. The **range** statements allow the table to access only records with the key value "Smith." The **get first** and **get next** statements read all of records from the range in the table. Note that the *same* key is used for all of the **range** and **get** statements.

```
{Clear any existing range for the table.}
range clear table Customers;

{Set up the new range.}
set 'Last Name' of table Customers to "Smith";
range start table Customers by number 1;
range end table Customers by number 1;

{Read the records in the range.}
get first table Customers by number 1;
while err() <> EOF do
  get next table Customers by number 1;
end while;
```

Chapter 4: Process-level Integrations

When you create process-level integrations, you will call procedures and functions in Microsoft Dynamics GP. This chapter contains information you need to use Microsoft Dynamics GP procedures and functions from pass-through sanScript. The following topics are discussed:

Procedures

A procedure is a script that provides functionality common to several parts of a Dexterity-based application. Procedures are called from other scripts in an application. When a procedure is called from another script, the two scripts involved have specific roles:

- The *calling* script is the script that accesses, or calls, the procedure. For Continuum, the pass-through sanScript is always the calling script.

- The *called* script is the procedure that is invoked by the calling script.

A set of parameters can be used to pass data to the called script or return data to the calling script.

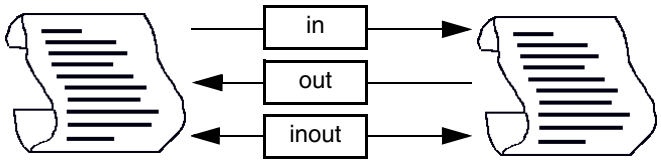
Calling procedures

The **call** statement temporarily transfers control to a procedure, passing an optional set of parameters. The syntax of the statement is shown below:

call {**background**} *procedure* {**of form** *form_name*}{, *parameter, parameter, ...*}

Parameters

Procedures operate by passing parameters between the calling script and the procedure. The procedure uses the information in these parameters to work with variables, fields and tables, and to return values to the calling script. Three types of parameters can be used, as shown in the following illustration.



Parameters are classified according to how they are used when the calling script communicates with the called script. The following table describes the different types:

Parameter type	Characteristics
in	The value is passed from the calling script to the called script. The called script can't change the value of an in parameter.
out	The value is passed from the called script back to the calling script. The called script sets the value of the out parameter.
inout	The value is passed from the calling script to the called script and then back to the calling script. The value can be used and changed by the called script, which then passes it back to the calling script.

Procedures allow up to 255 parameters to be passed between the calling script and the procedure. The types of the parameters defined for the procedure must match the types of the parameters passed from the calling script. For example, if the calling script passes an integer parameter to the procedure, the procedure must be set up to receive an integer as a parameter.

Typically, the parameters passed to and from a procedure have one of the standard storage types: boolean, currency, date, integer, long, string and time. Some procedures also use existing global field definitions to declare parameter types. This second method, described in the next section, is used when passing items such as composites to a procedure.

Using global fields as parameter types

In addition to the standard storage types, applications can also use existing global field definitions in a dictionary when declaring parameter types. This second method is used when items such as composites must be passed to a procedure.

The following example shows how a global field is used to specify a parameter type. The script calls the Verify_Account_Type procedure to verify account type information for a specified account. The procedure takes an Account Number as one of its parameters. The Account Number global field is used to specify the type of the account_number local variable.

```
{Account number}
local 'Account Number' account_number;

{Status flags}
local boolean account_status;
local boolean abnormal_termination;

{Set the segments of the account number.}
set component(1) of field account_number to "000";
set component(2) of field account_number to "1100";
set component(3) of field account_number to "00";

{Open the GL_Account_MSTR table and read the appropriate record.}
set 'Account Number' of table GL_Account_MSTR to account_number;
get table GL_Account_MSTR by number 6;
```

```
{Call the Verify_Account_Type procedure to verify that it is a Posting
account.}
call Verify_Account_Type, account_number,
➥  true,  {Posting account}
➥  false, {Unit account}
➥  false, {Posting allocation account}
➥  false, {Unit allocation account}
➥  false, {Fixed allocation account}
➥  false, {Variable allocation account}
➥  false, {Should the procedure read the account information?}
➥  account_status,
➥  abnormal_termination,
➥  table GL_Account_MSTR;
if abnormal_termination = false then
    if account_status = true then
        warning "Account is the specified type.";
    else
        warning "Account is not the specified type.";
    end if;
else
    error "Account type verification failed.";
end if;
```

Form procedures

Form procedures are functionally the same as standard procedures, except they're associated with a specific form. Form procedures are used to group procedures that perform tasks associated with a specific form in Microsoft Dynamics GP. Form procedures are called like standard procedures, except that the **of form** *form_name* clause must be included in the **call** statement.

Background processing

Background processing allows a procedure to run while other processing is occurring, such as opening windows, printing reports and so on. To run a procedure in the background, include the **background** keyword in the **call** statement. This will add the procedure to a processing queue that will process it when time is available. Procedures like posting are often run in the background.

Temporary tables can't be passed to a procedure running in the background because the temporary tables may no longer exist when the procedure is processed.

You can monitor the progress of background procedures by using the Process Monitor in Microsoft Dynamics GP. Background procedures must be completed before you exit Microsoft Dynamics GP. If you attempt to exit Microsoft Dynamics GP when a background procedure is running, a message will alert you that background processes are still running.

User-defined functions

Microsoft Dynamics GP contains several user-defined functions. You can use these user-defined functions in your pass-through sanScript the same way you use built-in functions such as **mktime()**.

Microsoft Dynamics GP also has form-level user-defined functions that are associated with a specific form. Like form procedures, they are used to group functions that perform tasks associated with a specific form. Form-level user-defined functions are called like standard functions, except that the **of form** *form_name* clause must be included immediately after the function call.

The following example shows how to call a form-level function. The script calls the **IsDupAccountNumber()** function of the GL_Copy/Move_Accounts form to verify whether an account already exists.

```
local 'Account Number' account_number;

{Set the segments of the account number.}
set component(1) of field account_number to "000";
set component(2) of field account_number to "1100";
set component(3) of field account_number to "00";

if IsDupAccountNumber(account_number) of form 'GL_Copy/Move_Accounts' = true
➥ then
    error "This account already exists.";
end if;
```

PART 2: SANSCRIPT REFERENCE

Part 2: sanScript Reference

This portion of the manual contains reference information for the sanScript language. The information is divided into the following chapters:

- Chapter 5, "Functions and Statements," contains descriptions of the sanScript functions and statements used for database-level and process-level integrations.
- Chapter 6, "Data types,"describes the various data types used in sanScript and provides information about how to work with them.
- Chapter 7, "Alert Messages," describes the compiler and runtime messages that can occur when you use pass-through sanScript.

Chapter 5: Functions and Statements

This chapter contains descriptions of the functions and statements used for process-level and database-level integrations. First, the functions and statements are listed in a table by their usage. Later, each is listed again in alphabetical order, with a detailed description, syntax, parameter list, and example of its use.

Category	Command	Description
Date	day()	Retrieves the day portion of a given date.
	month()	Retrieves the month portion of a given date.
	setdate()	Creates or modifies a date.
	sysdate()	Returns the current date from the current computer.
	year()	Retrieves the year portion of a given date.
Error handling	check error	Checks the value of the **err**() function for the last operation on the specified table and displays a message.
	err()	Returns the result of the last operation on a table.
Fields	clear field	Clears the specified field, or list of fields.
	fill	Sets the specified field or variable to its maximum value.
	set	Sets a field or variable to the value of an expression.
Messages	ask()	Displays a dialog box and returns a value indicating which button the user clicked.
	check error	Checks the value of the **err**() function for the last operation on the specified table and displays a message.
	error	Creates an error dialog displaying the specified string.
	warning	Creates a warning dialog displaying the specified string.
Numerics	str()	Converts a numeric value to a string.
	value()	Returns a numeric value corresponding to the first set of numbers encountered in the specified string.
Program structures	case...end case	Allows statements to run on a conditional basis.
	if then...end if	Runs a series of statements when a condition is met.
	repeat...until	Runs a series of statements until a condition is met.
	while do...end while	Runs a series of statements while a condition is met.
Script controls	abort script	Halts the current script.
	call	Starts the specified procedure.
Strings	str()	Converts a numeric value to a string value.
	value()	Derives a numeric value from a set of numeric characters in a string.
Tables	change	Reads a record and actively or passively locks it.
	clear table	Clears the specified table buffer.
	countrecords()	Counts the number of records in a table.
	get	Reads a record from a table without locking the record.
	range	Limits table access to a portion of the table.
	release table	Releases a locked or reserved record.
	remove	Removes a record or range from a table.
	save table	Saves the contents of the table buffer to a table.

Category	Command	Description
Time	**hour()**	Retrieves the hours portion of a given time.
	minute()	Retrieves the minutes portion of a given time.
	mktime()	Creates a time value from a given set of numbers.
	second()	Retrieves the seconds portion of a given time.
	systime()	Returns the current time from the current computer.

Command syntax

The syntax descriptions throughout this manual use the following standards:

- **Bold** text indicates language-specific reserved words, including both command names and words that appear literally in scripts.
- Any parentheses () are a part of the command.
- *Italics* indicate items to be replaced by object names or values.
- A vertical bar (|) between two or more items should be read as "or," and indicates that one item in the list should be chosen.
- Braces {} indicate optional items. If the braces enclose a group of items, one of the items can be chosen.
- Square brackets [] list a group of items in which one choice is required. In rare cases where they are part of the command syntax, they're set in bold text.
- Ellipses (...) indicate that other commands can appear between the keywords of the command being described.

Programming style

- Script examples are shown in Courier type.
- Each statement is terminated with a semicolon.
- Comments describing a line of the script appear above or beside the corresponding line and are enclosed in braces. A sample script statement with a comment is shown below:

```
{Release the lock on the current record.}
release table RM_Customer_MSTR;
```

- A continuation character (➥) indicates that a script continued from one line to the next in the manual should be typed as one line.

Command reference

The remainder of this chapter lists the sanScript commands that are typically used in pass-through sanScript for Continuum integrations.

abort script

Description The **abort script** statement stops the current script.

Syntax **abort script**

Parameters

- None

Comments The **abort script** statement is used to handle error conditions by stopping a script.

Example This script will be stopped if the values in the Debit and Credit variables aren't equal.

```
if Debit <> Credit then
    {Debit doesn't equal Credit. Stop the script.}
    abort script;
end if;
```

ask()

Description

The **ask()** function creates a dialog box that contains a message and up to three user-defined buttons. It returns a value that indicates which button is clicked by the user.

Syntax

ask(*prompt*, *button1*, *button2*, *button3***)**

Parameters

- *prompt* – A string that contains the message you want to display in the dialog box.
- *button1* – A string that contains the label for the first button in the dialog box.
- *button2* – A string that contains the label for the second button in the dialog box.
- *button3* – A string that contains the label for the third button in the dialog box.

Return value

An integer that indicates which button the user clicked. It corresponds to one of the following constants: ASKBUTTON1, ASKBUTTON2 or ASKBUTTON3.

Comments

The string value entered as the *prompt* parameter provides the message for the dialog box, and the buttons are labeled with the strings entered as the *button1*, *button2* and *button3* parameters. If you want to use fewer than three buttons, use a set of double quotation marks (`""`) for any buttons you don't want to use.

The window closes automatically after the user clicks a button.

Example

The following script asks the user whether the changes to the current customer record should be saved.

```
local integer answer;

answer = ask("Do you want to save changes?", "Yes", "Discard", Cancel,);

if answer = ASKBUTTON1 then
    {Yes was clicked.}
    save table RM_Customer_MSTR;
else
    {Discard was clicked.}
    release table RM_Customer_MSTR;
    clear table RM_Customer_MSTR;
end if;
```

Related items

Commands

error, **warning**

call

Description

The **call** statement transfers temporary control to a procedure.

Syntax

call {**background**} *procedure* {**of form** *form_name*}{, *parameter, parameter, ...*}

Parameters

- **background** – Specifies whether the procedure will be processed in the background. Certain Microsoft Dynamics GP procedures, such as posting, are performed in the background.
- *procedure* – The name of the procedure you wish to call.
- **of form** *form_name* – A parameter that must be included if the procedure is a form procedure. The *form_name* parameter is the name of the form the procedure is attached to.
- *parameter* – Values, such as tables or fields you wish to pass to or return from the procedure. Up to 255 parameters can be passed to or returned from the called procedure.

Comments

The **call** statement transfers temporary control to a procedure, passing information to it. If the **background** keyword is used, the called procedure is queued for execution in the background. If that background procedure calls another procedure, that procedure will also be run in the background. If the **background** keyword is not used, the called procedure can pass information back to the calling script.

You can't pass a table buffer when calling a procedure to run in the background because the pass-through sanScript might have finished running before the background procedure is run. In such a case, there would be no table buffer for the procedure to use. Similarly, you can't pass a temporary table to a background procedure, since the temporary table may no longer exist when the background procedure is processed.

You can monitor the process of procedures by using the Process Monitor in Microsoft Dynamics GP.

Example

In the following example, the Get_Next_Journal_Entry procedure is called to retrieve the next valid journal entry for a general ledger transaction.

```
local long journal_entry;
local boolean status;

call Get_Next_Journal_Entry, true, journal_entry, status;

if status = false
    warning "Unable to get a journal entry.";
end if;
```

case...end case

Description

The **case...end case** statement allows a series of statements to run on a conditional basis, much like the **if then...end if** statement.

Syntax

case *exp* **in [***value***]** *statements* {**in [***value***]** *statements*} {**else** *statements*} **end case**

Parameters

- *exp* – A field or expression that is to be compared to the *value* parameter on an equality basis.
- **in [***value***]** – The *value* can be a single value or a range of values that the *exp* parameter can be equal to. The value must be enclosed in brackets. Use the word "to" to express a value range, such as "1 to 10". The *value* for the expression "A or B" can be written as "A, B".
- *statements* – Any valid sanScript statement or group of statements.
- **else** – If the else clause is included, then the statements following it will be run if all of the **in []** clauses have been evaluated as false.

Comments

If multiple **in []** clauses are included, the statements after the first **in []** clause to be evaluated as true will be run, and subsequent clauses will not be evaluated.

If none of the **in []** clauses are evaluated to be true and an **else** clause isn't included, then no statements will be run until those listed after the **end case** statement.

Example

The following example uses the **case...end case** statement to set the rebate amount to be paid to customers based on their total year-to-date sales.

```
local currency rebate_amount;

case 'Total Sales YTD' of table RM_Customer_MSTR_SUM
    in [0 to 999]
    {The customer purchased less than $1,000 worth of goods.
    They aren't eligible for a rebate.}
    set rebate_amount to 0;
    in [1000 to 2999]
    {The customer purchased between $1,000 and $2,999 worth of
    goods. They qualify for a $15 rebate.}
    set rebate_amount to 15;
    in [3000 to 4999]
    {The customer purchased between $3,000 and $4,999 worth of
    goods. They qualify for a $25 rebate.}
    set rebate_amount to 25;
else
    {The customer purchased over $5,000 worth of goods. They
    qualify for a $50 rebate.}
    set rebate_amount to 50;
end case;
```

Related items

Commands
if then...end if

change

Description

The **change** statement reads a record from a table and passively or actively locks the record, allowing changes to be made to the table.

Syntax

change { **next** | **prev** | **first** | **last** } **table** *table_name* { **by** *key_name* | **by number** *expr* }{, **lock**}{, **wait**}{, **refresh**}

Parameters

- **next** | **prev** | **first** | **last** – Identifies which record to retrieve.

  If none of these keywords are included, the record that matches the key value in the table buffer will be retrieved. The **next** keyword will retrieve the record that follows the key value currently in the table buffer. The **prev** keyword will retrieve the record preceding the key value in the table buffer. The **first** keyword will retrieve the first record in the table or range. The **last** keyword will retrieve the last record in the table or range.

- **table** *table_name* – Identifies the table containing the record to read.

- **by** *key_name* | **by number** *expr* – Identifies the key by which you'll search the table to locate the record to be retrieved. If one of these parameters isn't included, the first key will be used.

  You can identify the key by its name or by the key number. In the **by** *key_name* parameter, *key_name* is the key's technical name. In the **by number** *expr* parameter, *expr* is an integer containing the number of the key, determined by its position in the table definition. For instance, the third key created can be identified by the number 3, and so on.

- **lock** – Indicates that the record will be actively locked, and no other users will be allowed to change or delete the record. If this option isn't included, the record will be passively locked.

Not all tables in Microsoft Dynamics GP support active locking. If you use the ***lock*** *keyword for a table that doesn't support active locking, you will receive a type incompatibility message when you compile the pass-through sanScript.*

- **refresh** – This keyword applies only for SQL tables. When a **change** statement reads a SQL table, a record is actually read from an in-memory buffer. The **refresh** keyword refreshes the records in the buffer before the read operation occurs, ensuring that the most-current data will be read.

Comments

The values of the key fields should be set in the table buffer before the **change** statement is issued. You don't need to set any fields in the table buffer to retrieve the first or the last record in a table.

You can use the **err()** function or the **check error** statement to handle any errors that may have occurred.

Example

The following example uses the **set** statement to copy a key value, Customer Number, from the parameter handler object. The **change** statement then uses the value in the table buffer to retrieve a record from the RM_Customer_MSTR table and place it in the table buffer. If the retrieval is successful, the credit limit amount for the customer is set to 50000 and the customer record is saved.

```
local boolean err_val;
local string cust_num;

{Retrieve the customer number from the parameter handler.}
set err_val to OLE_GetProperty("Customer Number", cust_num);
{Set the key value for the RM_Customer_MSTR table.}
set 'Customer Number' of table RM_Customer_MSTR to cust_num;
{Retrieve the record, allowing it to be changed.}
change table RM_Customer_MSTR;
if err() = OKAY then
    set 'Credit Limit Amount' of table RM_Customer_MSTR to 50000;
    {Save the changed record.}
    save table RM_Customer_MSTR;
    if err() <> OKAY then
        error "An error occurred while saving: " + str(err());
    end if;
end if;
```

Related items

Commands

check error, **err()**, **get**

Additional information

Chapter 3, “Database-level Integrations”

check error

Description

The **check error** statement checks the result of the last operation on the specified table, and displays a corresponding message if there was an error. If no table name is specified, it checks the result of the last table operation.

Syntax

check error {**table** *table_name*}

Parameters

- **table** *table_name* – An optional clause specifying the name of the table for which you want to check the last table operation.

Examples

In the following example, a record is being retrieved from the RM_Customer_MSTR table. The **check error** statement checks whether any errors occurred as the record was being retrieved. An appropriate alert message is displayed if an error occurred.

```
set 'Customer Number' of table RM_Customer_MSTR to "ADVANCED0001";
get table RM_Customer_MSTR;
{Display any message describing any error.}
check error;
```

In the following example, records are being retrieved from the RM_Customer_MSTR and RM_Customer_MSTR_SUM tables. The **check error** statements check whether any errors occur as the records are being retrieved from each table. It displays the appropriate alert message or messages if errors occur.

```
set 'Customer Number' of table RM_Customer_MSTR to "AARONFIT0001";
set 'Customer Number' of table RM_Customer_MSTR_SUM to "AARONFIT0001";
get table RM_Customer_MSTR;
get table RM_Customer_MSTR_SUM;
{Display any message describing any error.}
check error table RM_Customer_MSTR;
check error table RM_Customer_MSTR_SUM;
```

Related items

Commands
err()

clear field

Description

The **clear field** statement clears one or more fields.

Syntax

clear field *field_name*{, *field_name*, *field_name*, ...}

Parameters

- *field_name* – The name of the field to be cleared. If you have multiple fields to clear, list the name of each field separated by a comma.

Comments

Clearing a field removes the data currently in the field. Be sure to fully qualify the field or fields you are clearing.

If a table is not open when a field in that table is cleared, the **clear field** statement will open the table.

Example

The following example clears two fields in the table buffer for the RM_Customer_MSTR table.

```
clear field 'Customer Number' of table RM_Customer_MSTR,
➥ 'Customer Name' of table RM_Customer_MSTR;
```

clear table

Description The **clear table** statement clears a table buffer.

Syntax **clear table** *table_name*

Parameters

- *table_name* – The name of the table for which the table buffer should be cleared.

Comments Clearing a table removes the data currently in its table buffer. It doesn't remove data from the actual table.

Example The following example clears the table buffer for the RM_Customer_MSTR table.

```
clear table RM_Customer_MSTR;
```

countrecords()

Description The **countrecords()** function returns the number of records in the specified table.

Syntax **countrecords (table** *table_name***)**

Parameters

- **table** *table_name* – The name of the table you want to count records in.

Return value Long integer

Comments This function can also be used to retrieve an *estimate* of the number of records in a range.

Example The following example sets a local variable named customer_count to the number of records in the RM_Customer_MSTR table.

```
local long customer_count;

set customer_count to countrecords(table RM_Customer_MSTR);
```

day()

Description The **day()** function returns the day of the month in a given date.

Syntax **day(***date***)**

Parameters

- *date* – A date value.

Return value An integer between 1 and the maximum number of days in the month.

Example The following example sets a local variable named day_of_month to the number of the current day in the system date.

```
local integer day_of_month;

set day_of_month to day(sysdate());
```

Related items

Commands
month(), setdate(), sysdate(), year()

err()

Description

The **err()** function returns the result of the last operation on a specified table. If no table name is specified, it will return the result of the last table operation regardless of which table it was performed on.

Syntax

err({**table** *table_name*}**)**

Parameters

- **table** *table_name* – The name of table you want to check the last error for.

Return value

Integer

Comments

The **err()** function is often used in an **if...end if** structure that follows a table operation (**get**, **change**, **save table** and so on) to handle any errors that may have occurred during the operation. Using the **err()** function allows a script to detect errors, respond accordingly and specify whether to notify the user. In contrast, using the **check error** statement after table operations will automatically display an alert message notifying the user that an error occurred, but doesn't provide a method for the script to respond to the error.

Some of the return values from the **err()** function correspond to predefined constants, allowing you to use the constant in place of the error number. The following table lists common operation errors, and their associated error values and constants.

Constant	Value	Error type
OKAY	0	No Error
LOCKED	10	Locked Record
EOF	16	End of File
DUPLICATE	17	Duplicate Record
MISSING	18	Missing
RECORDCHANGED	30	Changed Record
DEADLOCKED	31	Deadlocked

The following table lists other operation errors and their associated error values. While no constants have been associated with these error values, they are valid values that can be returned by the **err()** function.

1 – Low on memory	22 – No table definition could be found
2 – Database manager not initialized	23 – Attempted to lock two records
3 – Database manager not supported	24 – No lock on update
4 – Too many tables opened	25 – Table doesn't match definition
5 – Record length too long	26 – The disk is full
6 – Too many keys for database type	27 – Unknown error
7 – Too many segments	28 – A non-modifiable key changed
8 – Table not registered	29 – Not a variable length field
9 – Table not found	32 – Path not found
11 – Table name error	33 – Buffer error
12 – Table not open	34 – Error in creating a Btrieve table
13 – Table not opened exclusive	35 – Invalid key definition
14 – Invalid command sent to database manager	36 – Maximum number of SQL connections reached.

15 – Key number doesn't exist	37 – Error accessing SQL data
19 – A set is already active	38 – Error converting SQL data
20 – Transaction in progress	39 – Error generating SQL data
21 – Not a variable length table	

Examples

The following script attempts to save the current record to the RM_Customer_MSTR table. The **err()** function is used to find out whether another user has changed the record. If the record has been changed, an error message is displayed indicating the situation to the user.

```
save table RM_Customer_MSTR;
if err() = RECORDCHANGED then
  error "This record was changed by another user.";
  release table RM_Customer_MSTR;
end if;
```

The following example retrieves each record from the RM_Customer_MSTR table and sets the Salesperson ID field to STEVE K. The **err()** function is used to check for any errors that occurred during processing.

```
{Attempt to read the first customer record in the table.}
change first table RM_Customer_MSTR;

while err() <> EOF do
    {Successfully read a customer record. Change the Salesperson ID.}
    set 'Salesperson ID' of table RM_Customer_MSTR to "STEVE K.";

    {Save the changed record.}
    save table RM_Customer_MSTR;

    {Check for any error.}
    if err() <> OKAY then
        error "Unable to update customer: " + 'Customer Number'
        ➥ of table RM_Customer_MSTR + " due to error " + str(err());
    end if;

    {Read the next record.}
    change next table RM_Customer_MSTR;
end while;
```

Related items

Commands

check error

error

Description

The **error** statement creates an error dialog box displaying the specified string. The dialog box will have one button labeled OK.

Syntax

error *expression*

Parameters

- *expression* – A string field, text field, or string or text value with the message to be displayed in the dialog box.

Example

The following example generates an error message for the user.

```
error "This is a test message.";
```

Related items

Commands
ask(), **warning**

fill

Description

The **fill** statement sets a field to the largest value represented by the field's data type, regardless of any keyable length or format applied to the field. For example, an integer field would be filled with the number 32,767.

Syntax

fill *field_name* {, *field_name*, *field_name*, ...}

Parameters

- *field_name* – The name of the field to be filled.

Comments

The **fill** statement is useful for setting ranges of information to be displayed from a table.

You can fill multiple fields using one **fill** statement, by listing each field name separated by a comma.

The following table lists the storage types for which the **fill** statement can be used, and the value with which the field will be filled:

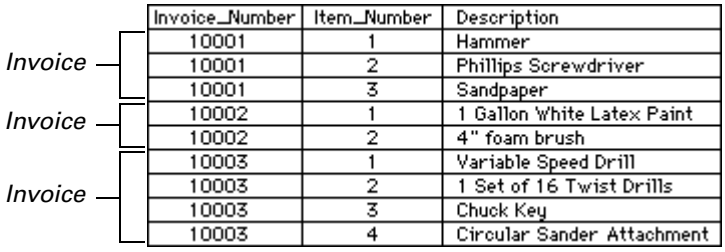
Storage type	Value
Date	12/31/9999
Currency	99999999999999.99999
Integer	32,767
Long	2,147,483,647
String	The length byte (first byte) of the string is set to the storage size of the string minus 1. Each of the remaining bytes is set to string equivalent of ASCII 255.
Time	23:59:59

Fields with date or time control types will be displayed using the format specified for their data types. For example, a filled time field might be displayed as 12:59:59 PM.

If the table containing the field to be filled is not open, the **fill** statement will open the table.

Example

The following example uses the **fill** statement to set the range for the Invoice table. The Invoice table is composed of records that contain the key fields Invoice_Number and Item_Number as shown in the following illustration:



Invoice_Number	Item_Number	Description
10001	1	Hammer
10001	2	Phillips Screwdriver
10001	3	Sandpaper
10002	1	1 Gallon White Latex Paint
10002	2	4" foam brush
10003	1	Variable Speed Drill
10003	2	1 Set of 16 Twist Drills
10003	3	Chuck Key
10003	4	Circular Sander Attachment

The number of items included in each invoice isn't known. The following script uses the **range** statement and the **fill** statement to set the range so only the items for Invoice 10002 will be accessed.

```
range clear table Invoice;
set Invoice_Number to 10002;
{Set the minimum value for Item_Number.}
clear Item_Number of table Invoice;
range start table Invoice;
{Set the maximum value for Item_Number.}
fill Item_Number of table Invoice;
range end table Invoice;
```

Related items

Commands
clear field, **range**

for do...end for

Description

The **for do...end for** statement runs a group of statements repetitively. The count will start at *expr1* and count up until a value equal to or greater than *expr2* has been reached. At each increment, all statements between the **for do** and **end for** keywords will be acted upon.

Syntax

for *index* **=** *expr1* **to** *expr2* {**by** *step_expr*} **do** *statements* **end for**

Parameters

- *index* – An integer variable used to keep track of the current number of repetitions of the loop. This variable for the loop index must be declared separately.
- *expr1* – An integer value that's the minimum value of *index*. The count will start at this number.
- *expr2* – An integer value that's the maximum value of *index*. The for loop will stop counting when *index* passes this number.
- **by** *step_expr* – The integer that specifies the amount by which the *index* variable will be incremented during each pass through the loop. Only positive step values are supported at this time. If this increment amount isn't stated, the loop will be incremented by one at each repetition.
- *statements* – any valid sanScript statement or statements.

Example

The following example sets the elements of the YTD_Sales array field to 0.

```
{Set up the loop index for the loop.}
local integer i;
local currency YTD_Sales[12];

for i = 1 to 12 do
    set YTD_Sales[i] to 0;
end for;
```

Related items

Commands

repeat...until, **while do...end while**

get

Description

The **get** statement retrieves a record from the table without locking the record. You can't change the data in a record that has been retrieved with the **get** statement.

Syntax

get { **next** | **prev** | **first** | **last** } **table** *table_name* { **by** *key_name* | **by number** *expr* } {,**refresh**}

Parameters

- **next** | **prev** | **first** | **last** – Identifies which record you want to retrieve. If none of these keywords are included, the record that matches the key value in the table buffer will be retrieved.
- **table** *table_name* – The name of the table that contains the record you want to read.
- **by** *key_name* | **by number** *expr* – Identifies the key by which you'll search the table.

  You can identify the key by its name or by its number. In the **by** *keyname* parameter, *keyname* is the key's technical name. In the **by number** *expr* parameter, *expr* is an integer containing the number of the key, determined by its position in the table definition. For instance, the third key created can be identified by the number 3, and so on. If one of these parameters isn't included, the first key will be used.

- **refresh** – This keyword applies only for SQL tables. When a **get** statement reads a SQL table, a record is actually read from an in-memory buffer. The **refresh** keyword refreshes the records in the buffer before the read operation occurs, ensuring that the most-current data will be read.

Comments

If the specified table isn't open, the **get** statement will open it.

The **err()** function or the **check error** statement can be used after the **get** statement to ascertain whether the operation was successful or to handle errors that occurred.

Example

The following example attempts to retrieve the customer name based on the customer number supplied in the parameter handler.

```
local string cust_num, cust_name;

{Retrieve the customer number from the parameter handler.}
set err_val to OLE_GetProperty("Customer Number", cust_num);

set 'Customer Number' of table RM_Customer_MSTR to cust_num;
get table RM_Customer_MSTR by number 1;
if err() = OKAY then
    set cust_name to 'Customer Name' of table RM_Customer_MSTR;
else
    error "An error occurred while reading the customer record: "+ str(err());
end if;
```

Related items

Commands

check error, **error**

Additional information

Chapter 3, "Database-level Integrations"

hour()

Description The **hour()** function returns the hours portion of a given time value.

Syntax **hour(***time***)**

Parameters

- *time* – A time value

Return value An integer between 0 and 23.

Example The following example sets the variable hour_of_time to the number of hours in the time value returned by the **systime()** function.

```
local integer hour_of_time;

set hour_of_time to hour(systime());
```

Related items

Commands
minute(), **mktime()**, **second()**, **systime()**

if then...end if

Description

The **if then...end if** statement allows statements to be run on a conditional basis.

Syntax

if *boolexp* **then** *statements* {**elseif** *boolexp* **then** *statements*}{**else** *statements*} **end if**

Parameters

- *boolexp* – Any expression that can be evaluated as true or false, such as:

```
A=B
Customer_Name="John Smith"
A+B<C
```

- **elseif** *boolexp* **then** – If one or more **elseif** clauses are included, and if the **if** clause has been evaluated as false, then the statements after the first **elseif** clause to be evaluated as true will be run.
- **else** – If the **else** keyword is included, statements following it will be run if the **if** clause and all **elseif** clauses have been evaluated as false. Only one **else** clause can be included in an **if then...end if** statement.
- *statements* – any valid sanScript statement or statements.

Examples

The following example displays a warning if the incorrect user is logged in to Microsoft Dynamics GP.

```
if 'User ID' of globals <> "SKUBIS" then
    warning "Incorrect user logged into Microsoft Dynamics GP.";
end if;
```

The following example reads through every item in the IV_Item_MSTR table and adjusts the list price based on the item type. The **if then**...**end if** statement selects the appropriate list price adjustment.

```
{Read the first inventory record.}
change first table IV_Item_MSTR;
while err() <> EOF do
    {Adjust the List Price.}
    if 'Item Type' of table IV_Item_MSTR = 1 then
        {Sales Inventory - 10% discount.}
        set 'List Price' of table IV_Item_MSTR to 0.9 *
        ➥ 'List Price' of table IV_Item_MSTR;
    elseif 'Item Type' of table IV_Item_MSTR = 2 then
        {Discontinued - 50% discount.}
        set 'List Price' of table IV_Item_MSTR to 0.5 *
        ➥ 'List Price' of table IV_Item_MSTR;
    elseif 'Item Type' of table IV_Item_MSTR = 5 then
        {Services - 25% discount.}
        set 'List Price' of table IV_Item_MSTR to 0.75 *
        ➥ 'List Price' of table IV_Item_MSTR;
    end if;
    save table IV_Item_MSTR;
    check error;
end while;
```

Related items

Commands

case...end case

minute()

Description The **minute()** function returns the minutes portion of a given time value.

Syntax **minute(***time***)**

Parameters

- *time* – A time value.

Return value An integer between 0 and 59.

Example The following example sets the variable minute_of_time to the number of minutes in the time returned by **systime()**.

```
local integer minute_of_time;

set minute_of_time to minute(systime());
```

Related items

Commands
hour(), **mktime()**, **second()**, **systime()**

mktime()

Description

The **mktime()** function creates a time value from three integer values.

Syntax

mktime(*hour*, *minute*, *second***)**

Parameters

- *hour* – An integer between 0 and 23 representing the number of hours.
- *minute* – An integer between 0 and 59 representing the number of minutes.
- *second* – An integer between 0 and 59 representing the number of seconds.

Return value

Time

Example

The following example sets a local variable named this_time to a time value of 2:25:37 PM.

```
local time this_time;

set this_time to mktime(14,25,37);
```

Related items

Commands
hour(), **minute()**, **second()**, **systime()**

month()

Description The **month()** function returns the month portion of a given date value.

Syntax **month(***date***)**

Parameters

- *date* – A date value.

Return value An integer between 1 and 12.

Example The following example sets a local variable named month_of_year to the number of the month in the date value returned by the **sysdate()** function.

```
local integer month_of_year;

set month_of_year to month(sysdate());
```

Related items

Commands
day(), setdate(), sysdate(), year()

range

Description

The **range** statement is used to select a portion of a table to use. A range can reduce the number of records that must be accessed in order to accomplish a task, increasing the speed and efficiency of your application.

Syntax

range [**start** | **end** | **clear**] **table** *table_name* { **by** *key_name* | **by number** *expr* } { **, inclusive** | **, exclusive** }

Parameters

- **start** | **end** | **clear** – Identifies the purpose of this particular range command. The **start** keyword sets the beginning of the range to the current values in the table buffer. The **end** keyword sets the end of the range to the current values in the table buffer. The **clear** keyword clears any range set for the key, but doesn't affect the table buffer.

- **table** *table_name* – The name of the table the range will be applied to.

- **by** *key_name* | **by number** *expr* – Identifies the key this range will be associated with. Keys can be identified by their name (**by** *key_name*) or by their numeric position in the table definition (**by number** *expr*). This parameter isn't used when the **clear** keyword is used. If no key is specified, the first key is used.

- **inclusive** | **exclusive** – In the **range end** statement, this keyword specifies how the range will be evaluated. This option applies only when the SQL database manager is being used for the table; it will be ignored for the other database managers.

  If the SQL database manager is being used, and the **inclusive** keyword is included, an *inclusive* range will be generated. If the **exclusive** keyword is included, a pure *exclusive* range will be generated. If a range type isn't specified, Dexterity will decide which type of range to use, based on how the key segments for the range have been set.

Comments

The selected range of the table will be treated as an entire table. For instance, a **get first** statement that includes the same **by** *key_name* or **by number** *expr* clause that the **range** statement used, will return the first record in the range.

Scope of the range

A range is associated with a key. The range will be used only when the table is accessed by the key with which the range is associated. All other keys will access the entire table.

You can define only one range at a time for a given table buffer, regardless of how many keys have been defined for the table. Each key can't *have its own range. To use a new range to access the table, you must clear the first range using the* ***range clear*** *statement.*

Clearing the range

A range is cleared by the **range clear** statement or closing the table. If you issue a **range** statement without first clearing an existing range, you'll be able to access data from the old range only.

Multisegment keys

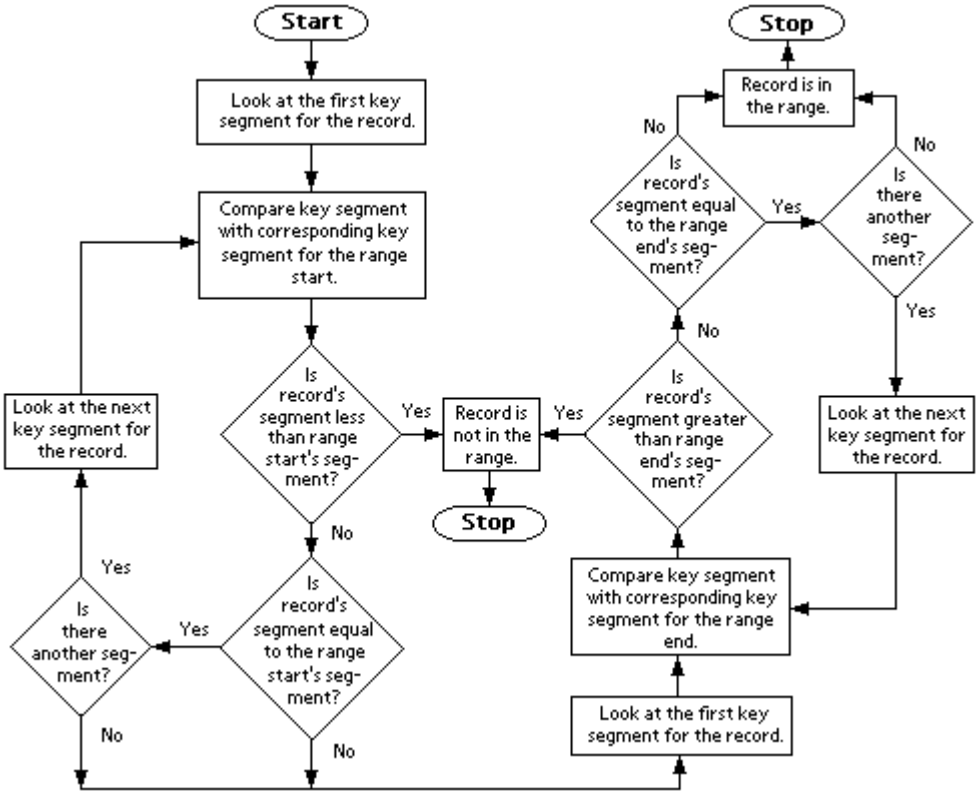
If a key is composed of several segments, you can create ranges based on several key segments. The **clear field** and **fill** statements are often used when setting ranges for multisegment keys. Typically, the first several corresponding key segments of the range start and range end are set to the same values. Then the remaining key segments are cleared and filled.

Evaluating the range

How a range is evaluated depends on the database manager used for the table, how the key segments for the table have been set, and whether the **inclusive** or **exclusive** keywords have been included in the **range end** statement.

For the c-tree database manager, ranges are always evaluated *inclusively.* Including the **inclusive** or **exclusive** keyword has no effect on the range.

The following flowchart describes the process used to evaluate an inclusive range.

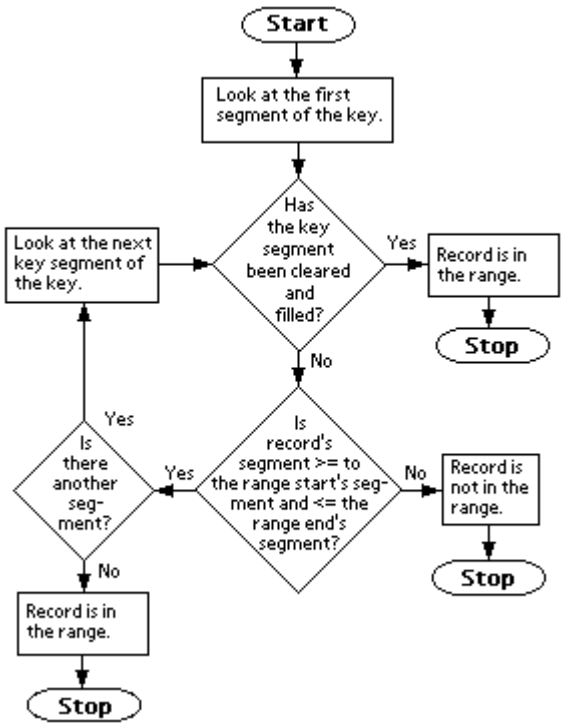


For the SQL database manager, how the range is evaluated depends on whether the **inclusive** or **exclusive** keyword is included in the **range end** statement. If the **inclusive** keyword is included, the range will be an inclusive range, just like the one produced for c-tree. If the **exclusive** keyword is included, the range will be evaluated *exclusively.*

The following flowchart describes the process used to evaluate an exclusive range.

*Evaluating inclusive ranges for SQL is **significantly** slower than evaluating exclusive ranges. For this reason, we recommend that you use inclusive ranges for SQL tables only when absolutely necessary.*

If the SQL database manager is used and neither the **inclusive** nor the **exclusive** keyword is included in the **range end** statement, Dexterity will determine the type of range used based on how the key segments for the range have been set. If the range is "well-behaved" or Dexterity can alter the range to make it "well-behaved," an exclusive range will be generated. Otherwise, an inclusive range will be generated.



Starting from the leftmost key segment and working to the right, a "well-behaved" range has the following characteristics:

1. The first 0 to *n* segments are set to equal values for both the range start and the range end.

2. The next 0 or 1 segments are set to non-equal values for the range start and range end.

3. The remaining segments (if any) are cleared for the range start and filled for the range end.

It is desirable for a range to be "well-behaved" because "well-behaved" ranges produce the same results for all database managers.

If Dexterity analyzes a range and finds the range to be "well-behaved" except that the rightmost key segments haven't been cleared and filled, those segments will be automatically cleared and filled and an exclusive range will be produced.

Records outside of the range

If you set a range and then set the key value out of the specified range and issue an unqualified **get** or **change** statement (one that doesn't include the **prev**, **next**, **first** or **last** keywords), you will receive an EOF error. However, depending on what value you set the key field to, you could successfully retrieve data by issuing a **get**, or **change** statement that uses the **next** or **prev** keywords. If you set the key value to a value *before* the range, then issue a **get next** statement, for example, the first record in the range will be retrieved. If you set the key value to a value *after* the specified range, then issue a **get prev** statement, the last record in the range will be retrieved.

Examples

The following example sets up a range so that only records with a customer name between "A" and "K" will be accessed. The second key for the table is used because it is composed of the Customer Name field. Note that the first line clears any previous range that may have been used before the new range is applied.

```
{Clear any existing ranges associated with this table.}
range clear table RM_Customer_MSTR;
set 'Customer Name' of table RM_Customer_MSTR to "A";
range start table RM_Customer_MSTR by number 2;
set 'Customer Name' of table RM_Customer_MSTR to "K";
range end table RM_Customer_MSTR by number 2;
```

The following example illustrates how to use a range for a multisegment key. The Purchase_Data table is shown in the following illustration. It has a key composed of the Purchase Date and the Store ID.

Purchase Date	Store ID	Amount
11/16/98	C	100.00
11/17/98	A	50.00
11/17/98	B	75.00
11/17/98	C	22.00
11/18/98	A	175.00
11/18/98	C	60.00
11/19/98	A	45.00
11/19/98	C	16.00
11/20/98	B	100.00

The following script sets a range to include all purchases made on 11/17/98 for all stores. The first segment of the key is set to the date 11/17/98. The second segment is set to its minimum value using the **clear field** statement. Then the start of the range is set. The first segment remains 11/17/98. The second segment is set to its maximum value using the **fill** statement. Then the end of the range is set. Using the **clear field** and **fill** statements on the Store ID fields allows all stores to be selected.

```
{Clear any previous range for the table.}
range clear table Purchase_Data;
{Set the start of the range.}
set 'Purchase Date' of table Purchase_Data to setdate('Purchase
➥ Date' of table Purchase_Data, 11, 17, 1998);
clear field 'Store ID' of table Purchase_Data;
range start table Purchase_Data by number 1;
{Set the end of the range.}
set 'Purchase Date' of table Purchase_Data to setdate('Purchase
➥ Date' of table Purchase_Data, 11, 17, 1998);
fill 'Store ID' of table Purchase_Data;
range end table Purchase_Data by number 1;
```

The following records are included in the range.

Purchase Date	Store ID	Amount
11/17/98	A	50.00
11/17/98	B	75.00
11/17/98	C	22.00

The following example illustrates how the **inclusive** keyword is included in the **range end** statement to force an inclusive range for a table with a SQL database type. The Purchase_Data table is shown in the following illustration. It has a key composed of the Purchase Date and the Store ID.

Purchase Date	Store ID	Amount
11/16/98	C	100.00
11/17/98	A	50.00
11/17/98	B	75.00
11/17/98	C	22.00
11/18/98	A	175.00
11/18/98	C	60.00
11/19/98	A	45.00
11/19/98	C	16.00
11/20/98	B	100.00

The following script sets a range to include all purchases from 11/17/98 to 11/19/98 where the Store ID is A.

```
{Clear any previous range for the table.}
range clear table Purchase_Data;
{Set the start of the range.}
set 'Purchase Date' of table Purchase_Data to setdate('Purchase
➥ Date' of table Purchase_Data, 11, 17, 1998);
set 'Store ID' of table Purchase_Data to "A";
range start table Purchase_Data by number 1;
{Set the end of the range.}
set 'Purchase Date' of table Purchase_Data to setdate('Purchase
➥ Date' of table Purchase_Data, 11, 19, 1998);
set 'Store ID' of table Purchase_Data to "A";
range end table Purchase_Data by number 1, inclusive;
```

The following records are part of the inclusive range.

Purchase Date	Store ID	Amount
11/17/98	A	50.00
11/17/98	B	75.00
11/17/98	C	22.00
11/18/98	A	175.00
11/18/98	C	60.00
11/19/98	A	45.00

Related items

Commands

fill, **remove**

Additional information

Ranges in Chapter 3, "Database-level Integrations"

release table

Description

The **release table** statement releases a lock on a record read with the **change** statement.

Syntax

release table *table_name*

Parameters

- *table_name* – The name of the table buffer that will have its current record released.

Comments

The table buffer won't be cleared when a **release table** statement is run. Use the **clear table** statement to clear the buffer.

Example

The following example releases the current record in the RM_Customer_MSTR table so another item can be read from or written to the table.

```
release table Customer_Master;
```

Related items

Commands

change

Additional information

Multiuser processing in Chapter 3, "Database-level Integrations"

remove

Description

The **remove** statement removes the current record or a range of records from the specified table.

Syntax

remove {**range**} **table** *table_name*

Parameters

- **range** – Causes the entire set of records that fall within the currently active range to be removed. If this keyword isn't used, only the record in the table buffer will be removed from the table.
- **table** *table_name* – The name of the table from which a record or range of records will be removed.

Comments

To remove a single record, the record must have been read using the **change** statement, because the record must be locked to be removed.

The record must be actively locked to guarantee that no other user in a multi-user system will be accessing the record when it's removed.

If the **remove range** statement is used for a table for which a range hasn't been set, all of the records in the table will be removed.

Examples

The following example removes the current record from the RM_Customer_MSTR table.

```
remove table RM_Customer_MSTR;
```

The following example removes records for all the customers from A to K for the RM_Customer_MSTR table. The range of customers to remove is set first, then the records are removed.

```
{Clear any existing ranges associated with this table.}
range clear table RM_Customer_MSTR;
set 'Customer Name' of table RM_Customer_MSTR to "A";
range start table RM_Customer_MSTR;
set 'Customer Name' of table RM_Customer_MSTR to "K";
range end table RM_Customer_MSTR;
remove range table RM_Customer_MSTR;
```

Related items

Commands

change, **range**

repeat...until

Description

The **repeat...until** statement is used to run statements repetitively. The statements enclosed in the repeat statement are run, then the exit condition is tested. If the condition returns a false value, the loop is continued. If true is returned, the loop is exited.

Syntax

repeat *statements* **until** *boolexp*

Parameters

- *statements* – Any valid sanScript statements.
- *boolexp* – Any expression that can be evaluated as true or false, such as:

```
A=B
Customer_Name="John Smith"
A+B<C.
```

Example

The following example reads all the records in the RM_Customer_MSTR table. Records are read until an End of File (EOF) error is returned.

```
get first table RM_Customer_MSTR;
if err() <> EOF then
    {Indicates there are records to retrieve.}
    repeat
        get next table RM_Customer_MSTR;
    until err() = EOF;
    {EOF signals the end of the table has been reached.}
end if;
```

Related items

Commands

for do...end for, while do...end while

save table

Description

The **save table** statement saves the current contents of the table buffer to the table.

Syntax

save table *table_name*

Parameters

- *table_name* – The name of the table that will have its table buffer contents saved.

Example

The following example retrieves each record from the RM_Customer_MSTR table and sets the Salesperson ID field to STEVE K. Then the record is saved.

```
{Attempt to read the first customer record in the table.}
change first table RM_Customer_MSTR;

while err() <> EOF do
    {Successfully read a customer record. Change the Salesperson ID.}
    set 'Salesperson ID' of table RM_Customer_MSTR to "STEVE K.";

    {Save the changed record.}
    save table RM_Customer_MSTR;

    {Check for any error.}
    if err() <> OKAY then
        error "Unable to update customer: " + 'Customer Number'
        ➥ of table RM_Customer_MSTR + " due to error " + str(err());
    end if;

    {Read the next record.}
    change next table RM_Customer_MSTR;
end while;
```

Related items

Additional information

Common table operations and *Multiuser processing* in Chapter 3, "Database-level Integrations"

second()

Description The **second()** function returns the seconds portion of a given time value.

Syntax **second(***time***)**

Parameters

- *time* – A time value.

Return value An integer between 0 and 59.

Example The following example sets the variable second_of_time to the number of seconds in the time value returned by the **systime()** function.

```
local integer second_of_time;

set second_of_time to second(systime());
```

Related items

Commands

hour(), **minute()**, **mktime()**, **systime()**

set

Description

The **set** statement assigns the value of an expression to a field or variable.

Syntax

set *item* **to** *expression*

Parameters

- *item* – The field or variable that's to be set to a value.
- *expression* – The value to be assigned to the field.

Comments

The field can have any data type, but the expression should have the same storage type as the field. If you use the **set** statement to set the value of a field in a table that isn't open, the **set** statement will open the table.

Examples

The following example shows the **set** command.

```
set 'Customer Name' of table RM_Customer_MSTR to "Ace Electric";
```

The following example shows the **set** command used to perform a calculation.

```
local currency discount_price;

set discount_price to 'List Price' of table IV_Item_MSTR * 0.75;
```

setdate()

Description

The **setdate()** function is used to create a date value or modify an existing date value.

Syntax

setdate(*date*, *month*, *day*, *year***)**

Parameters

- *date* – The date field or variable you want to modify.
- *month* – A new month value for the date, in the range 0 to 12. If set to 0, the current month value of the specified date field won't be modified. If an attempt is made to enter an out-of-range month, an alert message is automatically displayed to the user.
- *day* – A new day value for the date, in the range 0 to 31. If set to 0, the day value of *date* won't be modified. If an attempt is made to enter an out-of-range day for any month, an alert message is automatically displayed to the user.
- *year* – A new 4-digit year value for the date. If set to 0, the year portion of *date* won't be modified.

Return value

Date

Example

The following example sets the value of the start_date variable. The day is set to the 12th day of October, and the year is set to 1998.

```
local date start_date;

set start_date to setdate(start_date, 10, 12, 1998);
```

Related items

Commands

day(), **month()**, **sysdate()**, **year()**

str()

Description

The **str()** function returns a string representation of a numeric value. This is useful for any situation in which you want to include a number in a string to be displayed to the user.

Syntax

str(*expression***)**

Parameters

- *expression* – The numeric variable, field, or value you wish to convert to a string.

Return value

String

Comments

The **str()** function is commonly used to convert non-string values to string values so the parameter handler can be used to return them to the Continuum application.

Example

The following example converts the value of the List Price currency field to a string so it can be included in a message to the user.

```
warning "Current price is $" + str('List Price' of table RM_Customer_MSTR);
```

Related items

Commands

value()

Additional information

Chapter 2, “Working With Data”

sysdate()

Description The **sysdate()** function returns the current date from the current computer.

Syntax **sysdate()**

Parameters

- None

Return value Date

Example The following example sets the value of the start_date variable to the system date.

```
local date start_date;

set start_date to sysdate();
```

Related items

Commands

day(), month(), setdate(), year()

systime()

Description The **systime()** function returns the current system time from the current computer.

Syntax **systime()**

Parameters

- None

Return value Time

Example The following example sets the value of the start_time variable to the system time.

```
local time start_time;

set start_time to systime();
```

Related items

Commands
hour(), minute(), mktime(), second()

value()

Description

The **value()** function returns a numeric value containing the first set of numbers encountered in a specified string.

Syntax

value(*string***)**

Parameters

- *string* – The string expression or string field you wish to evaluate.

Return value

Initially, the value is returned as a currency type. It is converted to the appropriate numeric type based on the storage type of the field or variable the value is set to.

Comments

If the string value contains letters and numbers, only the first numbers in the string will be converted. The conversion process will begin with the first number encountered, and end when the first character that is not a number is encountered. For example, the string "Jones78Smith8" will be converted to 78.

If the string value contains a number with the system-defined decimal separator, the return value may be rounded, depending on the type of field the return value is set to. If the data type is set to an integer or long integer, the return value will be rounded up or down as appropriate. If the value is returned to a currency field or variable, the value won't be rounded.

If a string contains no numbers, a value of 0 will be returned.

The **value()** function is commonly used to convert string values to their numeric equivalents after the Continuum application has used the parameter handler to pass them to pass-through sanScript.

Examples

The following example converts the string "A123" to the value 123 and assigns the value to the integer variable new_integer.

```
local integer new_value;

set new_integer to value("A123");
```

The following example converts the string "Cost123.45" to the value 123 and assigns the value to the integer variable new_integer. The value is rounded because it is returned to an integer variable.

```
local integer new_value;

set new_integer to value("Cost123.45");
```

The following example converts the string "Cost123.45" to the currency value 123.45000 and assigns the value to the variable new_currency. The value isn't rounded because it is returned to a currency field.

```
local currency new_currency;

set new_currency to value("Cost123.45");
```

Related items

Commands

str()

Additional information

Chapter 2, "Working With Data"

warning

Description

The **warning** statement creates a warning dialog box displaying the specified string. The dialog box will have one button labeled OK.

Syntax

warning *expression*

Parameters

- *expression* – A string field, text field, or string or text value with the message to be displayed in the dialog box.

Example

The following example generates a warning message for the user.

```
warning "This is a test message.";
```

Related items

Commands
ask(), **error**

while do...end while

Description

The **while do...end while** statement runs statements repetitively. The statements enclosed in the **while do...end while** statement are run as long as a boolean expression remains true. The expression is evaluated once before each repetition of the loop.

Syntax

while *boolexp* **do** *statements* **end while**

Parameters

- *boolexp* – Any expression that can be evaluated as true or false, such as:

```
A=B
Customer_Name="John Smith"
A+B<C
```

- *statements* – Any valid sanScript statements.

Example

In the following example, items are read from the RM_Customer_MSTR table until the end of the table is reached.

```
get first table RM_Customer_MSTR;
while err() = OKAY do
    {While the end of the table hasn't been reached, read the next item.}
    get next table RM_Customer_MSTR;
end while;
```

Related items

Commands

for do...end for, **repeat...until**

year()

Description

The **year()** function returns the year portion of a given date value. The returned value will be a four-digit year, such as 1998.

Syntax

year(*date***)**

Parameters

- *date* – A date value.

Return value

Integer

Example

The following example sets a local variable named year to the number of the year in the date value returned by the **sysdate()** function.

```
local integer year;

set year to year(sysdate());
```

Related items

Commands
day(), **month()**, **setdate()**, **sysdate()**

Chapter 6: Data types

The following is a list of the data types used for fields that are stored in tables, and provides an example illustration and information of how to work with fields of each type.

Boolean

Example None

Description Stores a boolean (true or false value). The default value is false.

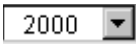
Check box

Example

Description Stores and displays a boolean (true or false) value. The value in the field is true if marked and false if unmarked.

Combo box

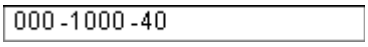
Example



Description Allows a text item to be entered by a user or chosen from the list. The value in the field is stored as a string.

Composite

Example



Description A composite is a special data type that is composed of several individual fields. The Microsoft Dynamics GP Account Number field is a composite.

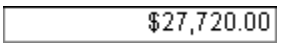
Each portion of the composite is called a component. To reference a component, use the sanScript **component** keyword, followed by the component number enclosed in parentheses. For example, the following sanScript code references the second component of the Microsoft Dynamics GP account number:

```
local string segment_2;

set segment_2 to str(component(2) of field 'Account Number' of
➥ table GL_Account_MSTR);
```

Currency

Example



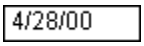
Description

Displays a value as a currency amount, with a currency symbol and thousands separator if specified in the data type's format.

The currency value can be in the range [-99,999,999,999,999.99999 to 99,999,999,999,999.99999]. The decimal point is implied in the number, but not actually stored. For display purposes, currency values are limited to 14 digits to the left of the decimal and 5 digits to the right. Any formatting, such as the currency symbol and thousands separator, is not stored with the value.

Date

Example



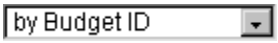
Description

Stores and displays a date. To set the value of a data field or variable, use the **setdate()** function. To read the day, month and year portions of a data value, use the **day()**, **month()** and **year()** functions. No formatting information is stored with the date value.

An uninitialized date field (one that hasn't been set to a value) will have the value 000000.

Drop-down list

Example

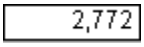


Description

The value of the drop-down list is an integer associated with the item selected in the list. Because the list items can be sorted, the value associated with an item may not correspond to its position. For example, the first item appearing in the drop-down list could have the value 3 associated with it. This means that when the item is selected, the drop-down list will have the value 3, not 1 as you might expect.

Integer

Example

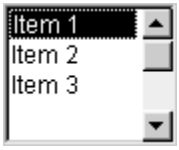


Description

Displays and stores integers from -32,767 to 32,767.

List box

Example

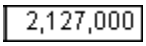


Description

The value of the list box is an integer associated with the item selected in the list. Because the list items can be sorted, the value associated with an item may not correspond to its position. For example, the first item appearing in the list could have the value 3 associated with it. This means that when the item is selected, the drop-down list will have the value 3, not 1 as you may expect.

Long integer

Example



Description

Displays and stores integers from -2,147,483,648 to 2,147,483,647.

Radio group

Example

Description

Groups radio buttons and stores a single integer value corresponding to the position of the selected radio button in the tab sequence. If the first radio button is selected, the value 0 is stored; if the second one is selected, the value 1 is stored, and so on.

String

Example

Description

Displays and stores strings of up to 255 characters.

Text

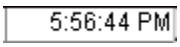
Example

Description

Displays and stores text up to 32,000 characters in length.

Time

Example



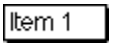
Description

Stores and displays a time value in 24-hour format. To set the value of a time field or variable, use the **mktime()** function. To read the hour, minute and second portions of a time value, use the **hour()**, **minute()** and **second()** functions. No formatting information is stored with the time value.

An uninitialized time field (one that hasn't been set to a value) will have the value 000000.

Visual switch

Example



Description

Displays a series of items. The value of the field is an integer corresponding to the position of the currently-displayed item in the series, starting with 1 and incremented by 1.

Chapter 7: Alert Messages

This chapter describes the alert messages that can occur when you use pass-through sanScript. It lists compiler messages and runtime messages.

Compiler messages

Compiler messages can occur when you send pass-through sanScript to the Microsoft Dynamics GP runtime engine to be compiled and executed. When a compiler error occurs, the message will be returned in the CompileErrorMessage parameter of the **ExecuteSanScript** method.

Some messages identify the resource, such as a field or form, that caused the error. Italic type is used to indicate items that will be substituted when the message is returned.

Array *array_name* must have a constant size.

Situation When creating a local array, you attempted to use a variable to set the size of the array. *Array_name* is the array that must have a constant size.

Solution Be sure you're using a constant value when setting the size of a local array.

Can't *message* a constant.

Situation You attempted to use a constant in a command that doesn't allow a constant as a parameter. For example, if you attempt to clear or fill a constant using the **clear** or **fill** statements, this message will appear since a constant isn't a valid parameter for these statements. *Message* is the string indicating the attempted action for the constant, such as fill or clear.

Solution Remove the reference to the constant, or use a statement with which the constant can be used as a valid parameter.

Cannot find *resource resource_name.*

Situation You've referenced a resource that can't be found. *Resource* is the type of resource that can't be found. *Resource_name* is the name of the resource that can't be located.

Solution Common solutions are:

- Check spelling. Be sure the resource name is spelled correctly in the script.
- Be sure the resource name is properly qualified. For example, if the resource is a global variable, be sure that you're using the qualifier "of globals" in the script.

Comment unterminated.

Situation You started a comment using an opening brace - { -, but didn't indicate the end of the comment with a closing brace - } -. There must be one closing brace for each opening brace.

Solution Be sure you've included both braces when adding comments to a script.

Expression not allowed in *function_name.*

Situation You've used a function that doesn't allow an expression as a parameter. *Function_name* is the name of the function that contains the invalid expression.

Solution Be sure that the function you're using allows an expression as a parameter before you attempt to compile the script.

Index out of bounds for array *array_name.*

Situation You've entered a value for the array index that's larger than the array size. For instance, if an array component is referenced as `Array[6]` in a script but the array has a size of 5, the error will occur. *Array_name* is the name of the array field.

Solution Use an array index within the upper bounds of the array field or the local variable definition.

Local variable *variable_name* not found.

Situation You've referenced a local variable that can't be found. *Variable_name* is the name of the local variable that can't be located.

Solution Common solutions are:

- Check spelling. Be sure the local variable name is spelled correctly where it's defined and where you refer to it in the script.
- Be sure the local variable has been defined.
- If the item shouldn't be a local variable, it's probably not properly qualified.

***Name* declared twice.**

Situation Two declared items, such as local variables, have the same name. *Name* is the name of the item that's declared twice.

Solution Change the name of one of the items.

***Field_name* is an array and requires a subscript.**

Situation You've referenced an array field, but didn't include the square brackets - [ ] - around the array index.

Solution Be sure to include the square brackets and array index when referencing an array.

***Field_name* is not an array.**

Situation You've referenced a field as an array, but the field isn't an array. *Field_name* is the name of the field you tried to reference.

Solution Change the script so the field isn't referenced as an array.

Operands incompatible with operator *operator.*

Situation You've used an operator with incompatible operands. For example, adding a string and a number together will cause this message to appear. *Operator* is the operator that can't be used with the operands.

Solution Be sure the operands are of the same type and are in the appropriate expression type (string, integer, boolean, and so on). Refer to Chapter 1, "sanScript," for more information.

Too many local variables.

Situation A script is using more than 64K of local variable space.

Solution Reduce the size of the local variables used to less than 64K. If necessary, divide the script into several smaller scripts.

Probably an infinite *loop_type* loop.

Situation The index variable for a "while" loop created using the **while do...end while** statement or a "repeat" loop created using the **repeat...until** statement isn't modified inside the loop body. When the index variable isn't modified, no exit condition exists and an infinite loop occurs. *Loop_type* indicates the type of loop that might be infinite.

Solution Be sure that an appropriate exit condition exists for the *boolexp* parameter specified for each of these statements. For the **while do...end while** statement, the *boolexp* must return false for the exit condition to occur. For the **repeat...until** statement, *boolexp* must return true for the exit condition to occur.

Syntax error, probably a missing end <something> or a missing ';'.

Situation The script compiler can't locate the end marker for a command or a line in a script.

Solution Review the statements in the script and be sure there's a semicolon at the end of each. Also be sure that each statement requiring an "end" keyword, such as **end if** or **end while**, has one.

Syntax error: *name.*

Situation The script compiler requires a certain keyword within a script command, but none is present. This renders other portions of the script command invalid.

Solution Common solutions are:

- Check keywords. Check the syntax of the commands that may have caused the error. If there are keywords missing for the command, such as the word "table" in the parameter "table *table_name*," be sure to add the proper keywords.
- Check spelling. Review any names that may be misspelled.
- Be sure you aren't using a keyword as a name.
- Review the statements in the script and be sure there's a semicolon at the end of each.
- Check for a missing "end if" portion of an **if then...end if** statement in the script. Be sure that you have an equal number of "if" and "end if" keywords if you use conditional statements in your script.

Table *table_name* does not have a key *key_name.*

Situation You've referenced a table key that can't be found. *Table_name* indicates the table for which the key *key_name* can't be located.

Solution Be sure the key name exists and is spelled correctly in the script.

The field *field_name* isn't in table *table_name.*

Situation A table field referenced in a script can't be found. *Field_name* is the name of the field that can't be located in the table *table_name.*

Solution Common solutions are:

- Be sure the name of the field is spelled correctly in the script.
- Be sure the field is part of the table. You can verify this by using the Table Descriptions window from the Resource Descriptions tool.

Type incompatibility *message.*

Situation The script compiler has found conflicting data types in a script. *Message* indicates the data types that are in conflict. This occurs if a value with one data type (such as an integer) is used in an expression for another data type (such as a boolean).

If the message was "active lock on non-active lock table," you have included the **lock** option in a **change** statement used to retrieve data from a table that does not allow active locking.

Solution If you're using an expression in the script, review the operands to be sure they use the same data type. If using a variable, be sure the data type for the variable matches the values set for that variable in script. If necessary, use the **str**() and the **value**() functions to convert data from one data type to another.

If the message was "active lock on non-active lock table," remove the **lock** option.

Type literal too large.

Situation The compiler has located a literal, or static value, that is too large. The type of the literal will be displayed in the message dialog box. This could be caused by a number that's too large to represent in a numeric data type, or a static string that's longer than the 255-character string limit.

Solution Be sure that the literal you're using has a valid length for the data type that it's being stored in.

Unknown identifier *name.*

Situation The script compiler can't locate the name of an item in the dictionary that you've referenced in script. *Name* is the item in the script that can't be identified.

Solution Review the name to be sure it's spelled properly.

Wrong number of arguments to *function_name.*

Situation A function you've used has an incorrect number of arguments (parameters). *Function_name* is the name of the function with the incorrect number of arguments.

Solution Review the parameters for the function to be sure you've used the correct number.

Runtime messages

Runtime messages can appear when your sanScript code is executed by the runtime engine. When a problem occurs, the message will be displayed in a dialog box in the application. Italic type is used to indicate items that will be substituted when the message is displayed.

Some messages will appear only if the **check error** statement is used in a script after a table operation.

A get/change next operation on table *table_name* has reached the end of the table.

Situation You attempted to read the next record in the table with the **get next** or **change next** statements, but the end of the table was reached. This message can appear when the **check error** statement is used. *Table_name* is the table that encountered the error.

Solution Use the **err**() function to check for the end of table (EOF) condition.

A get/change operation on table *table_name* caused a file sharing error.

Situation You attempted to read and actively lock a record that was actively locked by another user. This message can appear when the **check error** statement is used. *Table_name* is the table that encountered the error.

Solution Be sure no other users have placed an active lock on a record before you attempt to read and actively lock the same record. If multiple users must access the same record at the same time, use passive locking.

A get/change operation on table *table_name* could not find a record.

Situation You attempted to retrieve a record from a table, but the record couldn't be located. This message can appear when the **check error** statement is used. *Table_name* is the table that encountered the error.

Solution Check the key values to be sure they're correct. Also be sure data has been saved in the table before you retrieve records.

A get/change operation on table *table_name* failed. A record was already locked.

Situation You attempted to retrieve a record from a table, but there was already a locked record in the table buffer. *Table_name* is the table that encountered the error.

Solution To read another record from a table into the record buffer, the lock on the current record in the table must be released. Use the **release table** statement to release the lock from the current record before reading another record.

A get/change operation on table *table_name* is for an invalid key.

Situation You attempted to read a record from a table using a key that doesn't exist. *Table_name* is the table that encountered the error.

Solution Be sure that at least one table key has been defined for the table. Check whether you're using a key that doesn't exist.

A remove operation on table *table_name* caused a file sharing error.

Situation You attempted to remove a record from a table that was actively locked by another user. *Table_name* is the table that encountered the error.

Solution Be sure no other users or forms have placed an active lock on a record before you attempt to delete the same record. If multiple users must access the same record at the same time, use passive locking.

A remove operation on table *table_name* failed because the record couldn't be locked.

Situation You attempted to remove a record from a table, but the record was not locked. *Table_name* is the table that encountered the error.

Solution Be sure you've used the **change** statement to read the record and lock it before you use the **remove** statement to remove the record from the table.

A save operation on table *table_name* caused a file sharing error.

Situation The following situations can cause this message:

- You attempted to save a record that was actively locked by another form or another user. *Table_name* is the table that encountered the error.
- A form is accessing a table in read-only mode and is attempting to save to the table.

Solution Be sure no other users or forms have placed an active lock on a record before you attempt to save the same record. If multiple forms or users must access the same record at the same time, use passive locking.

A save operation on table *table_name* has created a duplicate key.

Situation You attempted to save a record that has the same key value as another record already in the table. This message can appear when the **check error** statement is used. *Table_name* is the table that encountered the error.

Solution Use a different command. This message will appear if you want to change the contents of records in a table, but are using the **get** statement instead of the **change** statement. If you want to change records in the table, be sure to use the **change** statement.

A save operation on table *table_name* record was changed by another user.

Situation You attempted to save a record, but another user had accessed the record and changed it. This message can appear when the **check error** statement is used. *Table_name* is the table that encountered the error.

Solution Read the record again to view the changes made by the other user. Then make appropriate changes to the information stored in the record and save it again.

An open operation on table *table_name* caused a file sharing error.

Situation You tried to open a table that was already opened for exclusive use. *Table_name* is the table that encountered the error.

Solution Wait until the other user is no longer accessing the table for exclusive use. Then attempt to open the table.

An open operation on table *table_name* failed accessing SQL data.

Situation You've attempted to retrieve data from a table.

Solution This alert message typically will include a More Info button. Click the More Info button to read the ODBC driver message, which will help you determine the source of the problem. The following list includes possible solutions:

- The database is full. Using your database's Administrator function, increase the size allocated for the database.
- The system log is full. Using your database's Administrator function, increase the size of the system log, back it up to tape, or dump (delete) it.

An open operation on table *table_name* failed because the maximum number of connections has been reached.

Situation You've attempted to perform an operation that requires a connection to a SQL data source, but the maximum number of connections allowed by your application has been reached.

Solution Wait for other operations that are using connections to be completed. You should then be able to complete the desired operation.

Bad power value.

Situation You attempted to use the power operator (^) with a base value other than 10.

Solution The power operator is supported only if the base value is 10. For example, 10^3 is a valid statement, while 2^3 is not.

Division by zero, script aborted.

Situation In a script, the divisor in a division operation was zero, causing the error.

Solution Determine whether the constant or variable used as the divisor in the division operation is zero or becomes zero during program operation.

Index *index_number* of *array_name* is out of range in *script_name.* Script terminated.

Situation The index value used to reference an array is larger than the size of the array. *Number* is the index value that is out of range. *Array_name* is the array for which the index value exceeded the size of the array. *Script_name* is the script that was active when the array size was exceeded.

Solution Be sure that any variable used as the index for the array doesn't exceed the size of the array. Determine whether the array is large enough or whether the array size was entered incorrectly when the array was created.

Long integer out of range. Results invalid.

Situation You've tried to store a long integer value or the result of a long integer expression in an integer field or variable. The resulting value is too large to be stored as an integer.

Solution Store the result of the expression in a long integer field or variable.

Background process is running: Exit aborted.

Situation A procedure was still running in the background when you tried to exit the application.

Solution Wait for the background task to finish before exiting.

String overflow during concatenation.

Situation A concatenation of two or more strings resulted in a string longer than 255 characters.

Solution Check the script where strings are concatenated (using the + operator) to determine whether the 255-character limit for strings is being exceeded.

String overflow on set *field_name.*

Situation A **set** statement attempted to fill a string field with a value of greater length than the field's keyable length.

Solution Check the keyable length of the target string and the number of characters in the string value of the **set** statement.

Glossary

Active locking
A method of locking that ensures only one user can change or delete the contents of a record at one time. The data in the locked record can't be changed or deleted by another user until the lock is released. The lock for the record is released when the user with the active lock moves to another record or closes the table.

Alert message
A message that appears when inappropriate, inadequate or unclear data or instructions are issued, when data is not accessible or when a confirmation is required.

Array
A field or variable containing multiple occurrences of the same type of information. The individual pieces of information stored by an array are called elements. For example, a seven-element array could be used to store daily sales totals instead of seven individual fields or variables.

Array index
The number designating a specific element within an array.

Ask dialog box
A modal dialog box generated by the **ask()** function. A dialog box displays a text string, an information icon and up to three push buttons allowing the user to make a selection.

Background processing
Processing, such as printing a report or running a procedure, that occurs while allowing other tasks to be completed simultaneously, such as entering data in a window.

Boolean expression
An expression that results in a value of true or false.

Buffer
A temporary storage area in a computer's memory.

c-tree Plus
A data manager used within the Microsoft Dynamics GP application.

Called script
The procedure or form procedure that's invoked by a calling script. The parameters for the called script are provided, or passed, by the calling script.

Calling script
The script that accesses, or calls, a procedure.

Check box
A data type that allow users to mark or unmark a selection. Check boxes are stored as boolean values.

Combo box
A data type that allow users to enter a text value or choose that value from a list. The combo box value is stored as a string.

Command
A function or statement included in the sanScript language.

Compile
To run a script through a compiler. A compiler translates the script instructions into a language that the computer can understand. Once the script has been compiled, the instructions within the script can be executed.

Compiler errors
Errors generated when a script is compiled.

Component
One field of a composite field.

Composite
A composite is a special data type that is composed of several individual fields. The Microsoft Dynamics GP Account Number field is a composite.

Constant
A fixed numeric or string value used in scripts. Several constants have been defined for Microsoft Dynamics GP. SanScript also has predefined constants that are used with specific functions or statements.

Data type
A resource in a Dexterity-based application that defines the characteristics for a field.

Date and time expression
An expression that results in either a date or time value. Date and time expressions can also result in a numeric value.

Deadlocked condition
The error condition that occurs when two users or scripts lock separate records and then also try to lock the records already locked by the other. For example, suppose script A locks record X and script B locks record Y. Script A then also attempts to lock record Y and script B also tries to lock record X. Both requests will be denied, forcing both scripts to wait endlessly for the desired record. Dexterity-based applications contain mechanisms to avoid deadlock conditions.

Dictionary
A group of resources that, when interpreted by the runtime engine, present a complete functioning application.

Drop-down list
A data type that allows users to select one item from a list. The value of a drop-down list is the integer associated with the selected item.

Element
One of the fields in an array field.

Error dialog box
The modal dialog box generated with the **error** statement. A text string, the standard error icon for the operating system, and an OK button are displayed in the dialog box.

Error trapping
To watch for and handle an exceptional event, such as an error. In sanScript, the **err()** function is used to trap table errors that occur at runtime, allowing the script to respond appropriately.

Expression
A sequence of operands and operators that are evaluated to return a value.

Field
A field contains a single piece of information used by the application dictionary. A field can be displayed in a window or stored in a table. The kind of information the field displays or stores depends on the data type associated with it.

Form
In Dexterity-based applications, a form is a collection of related windows, menus and scripts.

Form function
A user-defined function that's associated with a specific form.

Form procedure
A procedure that's associated with a specific form.

Function
A sanScript command that uses parameters and returns a value that must be used in an expression.

Global variable
A variable available to any script in the application at any time. Global variables are active the entire time a Dexterity-based application is open.

In parameter
A value that is passed from the calling script to the called script. The called script cannot change the value of an in parameter.

Inout parameter
A value passed from the calling script to the called script, then back to the calling script.

Key
A field or combination of fields within a record that is used as a basis by which to store, retrieve and sort records.

Key segment
One field of a group of fields that compose a key.

List box
A Dexterity data type that allows users to select one item from a list. The value of a list box is the integer associated with the selected item.

Local variable
A variable specific to a single script that is active only while the script is running. Local variables are defined at the beginning of the script.

Locking
The process of reserving the use of a record in a table. Locking can be passive, allowing others users to change or delete the record, or active, which doesn't allow other users to change or delete the record. A record must be locked before it can be changed or deleted.

Loop
A statement in script that runs repetitively until a defined condition is met. SanScript supports three loop structures: for, while and repeat.

Numeric expression
An expression that results in a numeric value.

Operand
An item in an expression that is acted on by an operator.

Operator
A symbol that indicates the action to perform on the operands in an expression.

Optimistic Concurrency Control
A form of record locking that enables several users to update the same record in a controlled manner, ensuring that record updates are completed properly and efficiently while placing minimal restrictions on the user's ability to access a record.

Order of precedence
The order in which the operations are carried out for an expression. The traditional order of precedence for numeric expressions is: first unary minus, then exponentiation, followed by multiplication and division, and finally addition and subtraction. This is also referred to as the order of evaluation.

Out parameter
The value passed from the called script back to the calling script.

Overflow
The condition that occurs in a numeric expression when an intermediate or final result of the expression is too large to be stored by the type of data used in the expression.

Pass
The process of sending data to and receiving data from a procedure by including a list of constants, variables or fields as parameters when the script is called.

Passive locking
A method of locking a record that allows other users to access and make changes to the record. The lock for the record is released when the user with access to the record moves to another record or closes the table.

Power operator
An operator symbolized in sanScript by a caret (^) that is supported in numeric expressions. The result is the first operand raised to the power of the second operand. In sanScript, only powers of 10 may be calculated.

Precedence
The order in which operations are performed for a type of expression.

Procedure
A script in a Dexterity-based application that can be called from other scripts to perform a common function.

Radio group
A data type in that's used to group related radio buttons and store the value of the selected button. A radio group's value is an integer corresponding to the selected radio button in the group.

Record
A collection of data made up of one instance of each field in a table.

Resource
An object such as a field, string, table, window or script that make up applications in Dexterity.

Resource Descriptions Tool
A tool that displays information about Microsoft Dynamics GP' fields, windows and tables.

sanScript
The scripting language in Dexterity.

Script
A list of instructions an application uses to perform tasks.

Statement
A script command used to complete a specific action in an application, such as saving a record to a table.

String
A sequence of up to 255 ASCII characters.

String expression
An expression that results in a string value.

Table
A collection of related data formatted in rows. Each row represents a separate record, and each column represents a separate field.

Table buffer
A buffer that acts as an intermediate storage area to hold one record from a table.

Table field
A field that's used to store information in tables.

Time expression
See *Date and time expression*.

Trap
To watch for and handle an exceptional event, such as an error. In sanScript, the **err()** function is used to trap table errors that occur at runtime, allowing the application to respond appropriately.

User-defined function
A script in a Dexterity-based application that you use in the same manner as sanScript's built-in functions.

Variable
A script item that allows an application to store values. They're called variables because their values can change. For example, a variable could be used to store the value returned from a dialog box created with the **ask()** function.

Warning dialog box
The modal dialog box generated with the **warning** statement. A text string, the standard warning icon for the operating system, and an OK button are displayed in the dialog box.

Index

R

S

T

U

V

W

Y