Microsoft®

Microsoft® Dexterity

sanScript Reference

Release 9.0

Publication date

October 2005

Contents

INTRODUCTION

Introduction

This manual provides a detailed reference for sanScript, the Microsoft® Dexterity scripting language. For general information about scripting, refer to Volume 2 of the Dexterity Programmer's Guide.

What's in this manual

The parts included in this manual are listed below, with a short description of each.

- Part 1, **SanScript Summary**, provides a summary table describing all of the functions and statements.
- Part 2, **Functions and Statements**, describes each sanScript command in detail.

Symbols and conventions

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

Symbol	Description
➥ `of table Items;`	A continuation character indicates that a script continued from one line to the next should be typed as one line in the Script Editor.
	The light bulb symbol indicates helpful tips, shortcuts and suggestions.
	Warnings indicate situations you should be aware of when completing tasks with Dexterity.
Margin notes summarize important information.	Margin notes call attention to critical information, and direct you to other areas of the documentation where a topic is explained.
SQL	The SQL symbol indicates information that applies only when a SQL database type is used.

Convention	Description
Part 2, **Basics**	Bold type indicates a part name.
Chapter 9, "Tables"	Quotation marks indicate a chapter name.
Applying formats	Italicized type indicates a section name.
`set 'l_Item' to 1;`	This font is used to indicate script examples.
RUNTIME.EXE	Words in uppercase indicate a file name.
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.

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:

```
{Open the form to enter customer data.}
open form 'Customer Maintenance';
```

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

PART 1: SANSCRIPT SUMMARY

Part 1: SanScript Summary

This part describes all of the functions and statements available in sanScript. The following table lists the functions and statements. The next part of the manual describes each function or statement in detail.

Application	
close application	Closes all windows and quits the current application.
run application	Starts an external application.

COM	
import	Creates a reference to the specified type library file, allowing the current script to access the items described in the type library.
new	Creates a reference to a new COM object.
this	Used to reference the current callback object from within a method defined for that callback object.

Commands	
check command	Sets a command to the checked state
disable command	Disables a command.
enable command	Enables a command.
hide command	Makes the named command invisible every place it is displayed,
run command	Performs the action for the specified command.
show command	Makes the named command visible every place it is displayed
uncheck command	Sets a command to the unchecked state
check command	Sets a command to the checked state

Currency	
currencydecimals()	Returns the current value of the operating system's decimal digits setting.
default roundmode to	Sets the default rounding behavior for calculations performed in the current script.
format()	Converts a currency or variable currency value to a string and applies a format.
precision()	Returns the total number of digits available for a variable currency field or variable.
round()	Returns a rounded value of a currency or variable currency field. The field can be rounded on either side of the decimal separator.
scale()	Returns the number of decimal digits for a variable currency field or variable.
set precision of	Sets the decimal precision of a currency or variable currency field.
truncate()	Returns a truncated value of a specified currency or variable currency field. The field can be truncated on either side of the decimal separator.

Date	
addmonth()	Adjusts the month portion of a given date.
day()	Retrieves the day portion of a given date.
dow()	Returns the day of the week for a given date.
eom()	Returns the last date of the month for a given date.
mkdate()	Creates a date value from three integer values.
month()	Retrieves the month portion of a given date.
setdate()	Creates or modifies a date.

Date (continued)	
sysdate()	Returns the current date from either the current computer or the server the computer is accessing.
year()	Retrieves the year portion of a given date.

Exception handling	
exit	Exits the current instance of the specified program structure.
restart script	Restarts the current script after an exception.
restart try	Restarts executing sanScript statements in a **try...end** try block after an exception.
throw	Throws a user exception or rethrows the current exception.
throw system exception for table	Throws a system exception for an error that has occurred for a table.
try...end try	Implements structured exception handling in an application.

Error handling	
assert	Allows verifying the state of computation at a given point in a script.
check error	Checks the value of the **err()** function for the last operation on the specified table and displays a message.
editexisting()	Indicates whether the last edit statement retrieved an existing record or reserved space for a new record.
err()	Returns the result of the last operation on a table.
error	Creates an error dialog box displaying a specified string. The dialog may also contain a Help button that will reference a specified entry in the help system.
naterr()	Returns the native error code for the last table operation on a specified table.
warning	Creates a warning dialog box displaying a specified string. The dialog may also contain a Help button that will reference a specified entry in the help system.

Fields and window objects	
add item	Adds a specified string to a specified list field. Can also add a numeric item associated with the specified string.
arraysize()	Retrieves the size of the specified array field or variable.
assign	Creates a reference to a window field.
change item	Changes the text for an item in a menu or a list field.
changed()	Returns a boolean value indicating whether any of the non-button fields in a window or form have been changed.
check field	Highlights a specified item in a multi-select list box or places a check mark next to an item in a button drop list.
checkedfield()	Indicates whether a specified position in a multi-select list box is selected or an item in a button drop list is checked.
clear field	Clears the specified field, or list of fields.
copy from field to field	Copies matching components from one composite to another.
countitems()	Returns the number of items in a menu or list field.
currentcomponent()	Returns an integer indicating which component of a composite that focus has moved from.

Fields and window objects (continued)	
datatype()	Returns the data type associated with a field.
default window to	Provides default qualification for window names within a script.
delete item	Removes an item from a list field or visual switch.
diff()	Returns the difference between a field's previous value and its current value.
disable field	Disables a field or multiple fields.
disable item	Disables a specific item in a button drop list.
empty()	Indicates whether a given field is empty.
enable field	Enables a field or multiple fields.
enable item	Enables a specific item in a button drop list.
fill	Sets the specified field to the maximum value each field's data type will allow.
filled()	Returns true if the specified field is set to its maximum value.
finddata()	Returns the numeric position of a data item in a list field that is associated with the specified numeric value.
finditem()	Returns the numeric position of a specific item in a menu or list field.
focus	Moves the input focus to a field.
format()	Converts a currency or variable currency value to a string and applies a format.
hide field	Makes the named field or fields invisible and inaccessible to the user.
insert item	Inserts an item into the specified position in a list field.
itemdata()	Returns the numeric value associated with the item in the specified position in the specified list field.
itemname()	Returns the text value in a specified position in a menu or list field.
length()	Returns the length of a specified string, string field or text field.
lock	Prevents a user from changing the contents of the specified field or fields.
move field	Changes the position of a field in a window.
old()	Retrieves the previous contents of a field after it has been changed.
override component	Sets the length of a component of an extended composite.
override field	Sets the separator for a composite and controls whether scroll arrows are displayed.
redraw	Updates and redisplays the specified field or fields.
resize field	Changes the display size of a specified field.
restart field	Restarts processing on a field.
return	Returns a value to a field in a different form that has requested a value.
set	Sets a field or variable to the value of an expression.
set precision of	Sets the decimal precision of a currency field.
show field	Makes the specified field or fields visible and accessible to the user.
uncheck field	Removes the highlight from a specified item in a multi-select list box or unchecks an item in a button drop list.
unlock	Allows a user to access previously locked fields.

Files	
getfile()	Allows the user to select a file from the operating system.
savefile()	Allows the user to name and save an export file.

Forms	
changed()	Indicates whether the contents of a form or window have changed.
clear changes	Clears the change flag for a window or form.
clear form	Clears the specified form.
close form	Closes the specified form.
default form to	Provides default qualification for form names within a script.
force changes	Sets the change flag for the specified window or form.
isopen()	Indicates whether a specified form or window is open.
open form	Opens the specified form.
open form with name	Opens a form for which the name is specified at runtime.
required()	Indicates whether all the required fields in a form or window contain values.
restart form	Clears a form and places the focus in the first field of the main window of the form.
return	Returns a value to a field in a different form that has requested a value.

Macros	
run macro	Runs a Dexterity macro.

Menus	
check menu	Places a check mark next to a specified menu item.
checkedmenu()	Indicates whether a specified menu item has a check mark by it.
countitems()	Returns the number of items in a menu or list field.
disable menu	Disables a menu item.
enable menu	Enables a menu item.
finditem()	Returns the numeric position of a specific item in a menu or list field.
hide menu	Hides the specified menu item.
itemname()	Returns the text value in a specified position in a menu or list field.
show menu	Shows the specified menu item.
uncheck menu	Removes a check mark from next to a menu item.

Messages and prompts	
ask()	Displays a dialog box and returns a value indicating which button the user clicked.
beep	Generates one of three sounds.
check error	Checks the value of the **err()** function for the last operation on the specified table and displays a message.
debug	Displays messages in Dexterity test mode.
error	Creates an error dialog box displaying the specified string.
getfile()	Allows the user to select a file from the operating system.
getmsg()	Retrieves the message string associated with a specified message ID.
getstring()	Creates a dialog box and returns a string entered by the user.
savefile()	Displays a dialog that allows the user to specify a file name and location for an export file.

Messages and prompts	
substitute	Substitutes text items for replacement markers in a string.
warning	Creates a warning dialog box displaying the specified string.

Numerics	
abs()	Returns the absolute value of an expression.
bitclear()	Clears a specified bit and returns the value.
bitset()	Sets a specified bit and returns the value.
bittest()	Tests the status of a bit.
decrement	Allows you to decrease a numeric value by a specified amount.
diff()	Returns the difference between a field's new value and its old value.
hex()	Returns the hexadecimal equivalent of an integral value.
increment	Allows you to increase a numeric value by a specified amount.
max()	Returns the greater of two values.
min()	Returns the lesser of two values.
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.

Process groups	
begingroup	Indicates the beginning of a group of procedures that are to be processed as a single entity. Also specifies details of how the group should be processed.
endgroup	Indicates the end of a process group defined by the **begingroup** statement.

Program structures	
case...end case	Allows a series of statements to run on a conditional basis.
continue	Continues the current instance of the specified loop type.
exit	Exits the current instance of the specified program structure.
for do...end for	Repeats a series of statements a given number of times.
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.

Reports	
run report	Prepares a predetermined report to be printed.
run report with name	Prepares a report, specified when the application is running, to be printed.

Resources	
resourceid()	Returns the resource ID for the specified resource.
technicalname()	Returns the technical name for the specified resource.

Script controls	
abort script	Halts the current script.
call	Starts the specified procedure.
call sproc	Starts a stored procedure on the current data source.
call with name	Starts a procedure that is specified at runtime.
clear force change	Stops a field's change script from automatically running once the force change statement has been issued for that field.
debugger stop	Opens the Script Debugger window and causes the current script to stop processing at that point.
execute()	Compiles and executes sanScript code at runtime.
force change	Runs the change script for a field when the focus leaves the field, regardless of whether the field's contents have changed.
missing()	Ascertains whether an optional parameter has been supplied to a procedure or function.
run script	Runs the change script for a given field.
run script delayed	Starts a script when other script processing is completed.

Scrolling windows	
delete line	Runs the line delete script for the scrolling window.
expand window	Switches between normal and expanded scrolling window line sizes.
fill window	Fills the scrolling window with data from a specified table.
hide window	Makes the specified scrolling window invisible and inaccessible to the user.
insert line	Runs the line insert script for the scrolling window.
move window	Changes the position of the specified scrolling window.
reject record	Skips specified records during fill and scrolling processes.
show window	Makes the specified scrolling window visible and accessible to the user.

Sound	
beep	Generates one of three sounds.

Strings	
ascii()	Returns the ASCII value of the first character in a string.
char()	Returns a one-character string corresponding to the specified ASCII value.
isalpha()	Indicates whether a string expression contains only alphabetic characters.
length()	Returns the length of a specified string, string field or text field.
lower()	Returns the specified string in lower case.
pad()	Adds a specified string to the beginning, end, or both the beginning and end of another specified string.
pos()	Returns the starting position of a given string within another text or string field.
replace()	Replaces a portion of a given string with another string.
str()	Converts a numeric value to a string value.
substitute	Substitutes values for replacement markers in a string.

Strings	
substring()	Retrieves a portion of a given string.
trim()	Removes a specified string from the beginning, end, or both the beginning and end of another specified string.
upper()	Returns the specified string in upper case.
value()	Derives a numeric value from a set of numeric characters in a string.

System calls	
currencydecimals()	Returns the current value of the operating system's decimal digits setting.
extern	Calls a function in a Dynamic Link Library (DLL).
physicalname()	Returns the physical name of a global field, the physical name of a table, or the column name of a field in a SQL table.
resourceid()	Returns the resource ID for the specified resource.
sysdate()	Returns the current date from either the current computer or the server the computer is accessing.
sysdatetime()	Returns the current date and time from the current computer.
system	Calls a Dexterity system command.
systime()	Returns the current time from either the current computer or the server the computer is accessing.
technicalname()	Returns the technical name for the specified resource.

Tables	
assign	Creates a reference to a table.
change	Reads a record and actively or passively locks it.
clear table	Clears the table buffer for the specified table.
close table	Closes the specified table.
column()	Retrieves the value from a specified field in a table.
copy from table	Copies fields from the table buffer to the window buffer.
copy from table to table	Copies matching fields from one table buffer to another.
copy from table to window	Copies information from the specified table buffer to the specified window buffer.
copy from window to table	Copies information from the specified window buffer to the specified table buffer.
copy to table	Copies fields from the window buffer to the table buffer.
countrecords()	Counts the number of records in a table.
delete table	Removes the named table from the operating system. When using the SQL database type, delete table can remove the table's data, but leave the table structure.
edit table	Reads or reserves a record from a table.
editexisting()	Indicates whether the last edit statement retrieved an existing record or reserved space for a new record.
fill table	Sets every field in the table buffer for the specified table to the largest value represented by the field's data type.
get	Reads a record from a table without locking the record.
keynumber()	Returns the number of a named table key.
open table	Opens the specified table using the specified options.

Tables	
range	Limits table access to a portion of the table.
range copy	Copies records in the current range from a source table to a destination table.
range where	Allows you to apply additional restrictions to a range of records in a 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.
sum range	Totals each field that has a numeric data type (integer, long integer or currency) in a specified range.

Time	
hour()	Retrieves the hour portion of a given time.
minute()	Retrieves the minute 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 either the current computer or the server the computer is accessing.

Transactions	
havetransactions()	Indicates whether a database type has transaction capability.
transaction begin	Starts a set of table operations.
transaction commit	Ends a set of table operations.
transaction rollback	Discards transaction results.

Triggers	
anonymous()	Used to refer to resources when registering triggers.
column()	Retrieves the value from a specified field in a table.
reject record	Stops processing of trigger processing scripts.

Windows	
abort close	Halts the process of closing a window.
changed()	Indicates whether the contents of a form or window have changed.
clear changes	Clears the change flag for a window or form.
clear window	Clears the specified window.
close palettes	Closes all palettes.
close window	Closes the specified window.
close windows	Closes all standard windows and palettes.
copy from window to window	Copies the contents of matching fields from one window to another.
force changes	Sets the change flag for the specified window or form.
isopen()	Indicates whether a specified form or window is open.
move window	Changes the position of the specified window.
open window	Opens the specified window.

Windows	
required()	Indicates whether all the required fields in a form or window contain values.
resize window	Changes the size of a window.
restart window	Clears a window and places the focus in the first field of the window.
set title of window to	Changes the display name of a window at runtime.

PART 2: FUNCTIONS AND STATEMENTS

Part 2: Functions and Statements

This part contains descriptions of all functions and statements available in sanScript. Each is listed in alphabetical order, with a detailed description, syntax, parameter list and example of its use.

abort close

Description

The **abort close** statement stops the process of closing a window.

Syntax

abort close

Parameters

- None

Comments

The **abort close** statement should be used only in window post scripts. The remainder of the script will continue to run even though the process of closing the window has been stopped.

Examples

The following script prevents a transaction window from being closed and displays a warning message if the transaction doesn't balance. Debits and Credits are window fields containing the sums of the debit and credit fields for the window.

```
if Debits <> Credits then
    {Stop the window from closing.}
    abort close;
    warning "Transaction does not balance.";
end if;
```

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.

Examples

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

```
if 'Debit' <> 'Credit' then
    {Script stops at this point only if Debit doesn't equal Credit.}
    abort script;
end if;
```

abs()

Description The **abs()** function returns the absolute value of the expression passed to it.

Syntax **abs(***numeric_expression***)**

Parameters

- *numeric_expression* – A numeric expression whose absolute value will be returned.

Return value The absolute value of the expression. If the expression type can be determined at compile time, that expression type will be returned. Otherwise, a variable currency value will be returned.

Examples The following example returns the absolute value of the expression 100 - 150.

```
local integer result;

result = abs(100 - 150);
```

add item

Description

The **add item** statement adds an item to a list field: a list box, multi-select list box, combo box, drop-down list, button drop list or visual switch. This statement also allows you to associate a numeric value with each item added to the list.

Syntax

add item *string_expression* {, *value*} **to** {**field**} *window_field*

Parameters

- *string_expression* – The string you want to add to the list field.
- *value* – An optional parameter containing the long integer to add to the list field in conjunction with the *string_expression*. If this parameter isn't used, 0 will be associated with the *string_expression*.
- **field** – An optional keyword identifying *window_field* as a field.
- *window_field* – A list field displayed in the window.

Comments

The **add item** statement adds the contents of the *string_expression* to a list field. The new item is added to the end of the list of items to be displayed. Items and their associated values are added at runtime, and aren't saved in the application dictionary. If items are added while the field is displayed, the field must be redrawn before the added string will appear.

The numeric values related to the added items won't appear in the list. However, they can be retrieved using the **itemdata()** function, and then used in scripts.

Examples

The following is part of a window pre script. It adds the items "Credit Card" and "On Account" to the Payment Method list box. The values "3" and "5" are also added, representing the surcharge percentage associated with these payment methods.

```
add item "Credit Card", 3 to field 'Payment Method';
add item "On Account", 5 to field 'Payment Method';
```

Related items

Commands

change item, **delete item**, **finddata()**, **finditem()**, **insert item**, **itemname()**, **redraw**, **Field_GetInsertPosFromVisualPos()**, **Field_GetVisualPosFromInsertPos()**

addmonth()

Description

The **addmonth()** function allows you to adjust the month portion of a date by a specified amount.

Syntax

addmonth(*date, months***)**

Parameters

- *date* – The date or datetime value you want to modify.
- *months* – The number of months you wish to adjust the month portion of the date by.

Comments

If the day value of the date is more than the number of days in the new month, then the day value of the date will be reduced by as much as is necessary to bring it back into range. For instance, if one month is added to the date 1/31/93, the value returned is 2/28/93, not 2/31/93.

Return value

Date

Examples

The following example adds three months to a date field named Working Month.

```
'Working Month' = addmonth('Working Month', 3);
```

Related items

Commands

day(), **dow()**, **eom()**, **mkdate()**, **month()**, **setdate()**, **sysdate()**, **year()**

anonymous()

Description

The **anonymous()** function is used when registering object triggers. It allows you to refer to a resource without requiring that the resource be open.

Syntax

anonymous (*resource_name***)**

Parameters

- *resource_name* – A fully qualified name denoting the field, form, window, table or menu to reference.

Return value

An anonymous reference to the specified resource.

Comments

The **anonymous()** function can only be used as a parameter for another statement or function, or when passing a reference to a table buffer to a procedure. You will receive a compiler error if you try to use this function in any other capacity.

To ensure proper registration, we suggest that all triggers be registered using a startup procedure that will run prior to the main dictionary's Main Menu form pre script. However, since this startup script is run before any windows in main dictionary are actually opened, resources named in the startup script won't be open, and in some instances may not exist. To avoid any problems this may cause, use the **anonymous()** function to refer to these resources.

Examples

This example registers a focus trigger used to check required fields in a form when the user presses the Save button. The IG_Check_Required_Fields procedure runs in response to this trigger:

```
local integer l_result;

l_result = Trigger_RegisterFocus(anonymous('Save Button'  of window
➥ RM_Customer_Maintenance of form  RM_Customer_Maintenance),
➥ TRIGGER_FOCUS_CHANGE, TRIGGER_BEFORE_ORIGINAL, script
➥ IG_Check_Required_Fields);
{Any result other than SY_NOERR means an error occurred.}
if l_result <> SY_NOERR then
    warning "Focus trigger registration failed.";
end if;
```

Related items

Additional information

Part 3, **Object Triggers**, in the Integration Guide

arraysize()

Description

The **arraysize()** function retrieves the size of the specified array field or variable.

Syntax

arraysize(*array_element***)**

Parameters

- *array_element* – Any element of an array field or variable.

Return value

An integer containing the size of the array field.

Examples

The following example retrieves the size of the Warning_List array. Any element of the array can be passed to the function.

```
local integer size;

size = arraysize(Warning_List[1]);
```

ascii()

Description

The **ascii()** function returns an integer corresponding to the ASCII value of a given character.

Syntax

ascii(*string***)**

Parameters

- *string* – The character value for which you want the ASCII value. If *string* contains more than one character, the ASCII value of only the first character will be returned.

Return value

Integer

Examples

The following script shows two different ways to set the local integer field ascii_value to the ASCII value of the letter "g".

```
ascii_value = ascii("g");
ascii_value = ascii("garage");
```

The following script sets the ascii_value field to the ASCII value of the first character of the passed-in string field, Customer_Name.

```
ascii_value = ascii(Customer_Name);
```

Related items

Commands

char()

Additional information

Appendix C, "ASCII Character Codes," in Volume 2 of the Dexterity Programmer's Guide

ask()

Description

The **ask()** function creates a dialog box containing a message and up to three user-defined buttons. It returns a value indicating which button is clicked by the user. The dialog box may also contain a Help button that will reference a specified entry in the help system.

Syntax

ask(*prompt*, *button1*, *button2*, *button3* {, *context_number*}**)**

Parameters

- *prompt* – A string field, text field or string or text value with the message to be displayed in the dialog box.
- *button1* – A string containing the label for the first button in the dialog box.
- *button2* – A string containing the label for the second button in the dialog box.
- *button3* – A string containing the label for the third button in the dialog box.
- *context_number* – A long integer specifying a help context number associated with a specific topic in the online help file for the current dictionary. If this parameter is used, a Help button will appear in the dialog box. If a user presses the Help button, the specified help file topic will be displayed. Refer to Chapter 7, "Windows Help," in the Dexterity Stand-alone Application Guide for more information.

Return value

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

Comments

If you want to use fewer than three buttons in the dialog box, use the empty string (`""`) for the buttons you don't want to use. For example, to display only two buttons, supply the button text for the first two buttons and the empty string for the third button.

The window closes automatically after the user clicks a button.

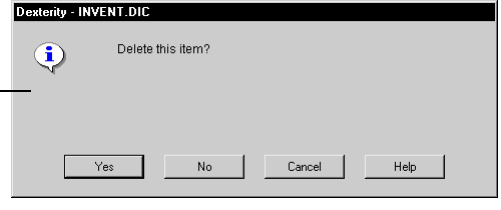
Examples

The following script is used in a transaction window to verify whether a record should be deleted. Note that a help context number (represented by the constant billion + 45) has been added. If the user clicks Help, the topic associated with that number will be displayed.

```
local integer answer;

answer = ask("Delete this item?","Yes","No","Cancel", billion + 45);
if answer = ASKBUTTON1 then
    {Yes was clicked.}
    remove table Item_Table;
    close form Item_Form;
elseif answer = ASKBUTTON2 then
    {No was clicked.}
    close form Item_Form;
else
    {Cancel was clicked.}
    abort script;
end if;
```

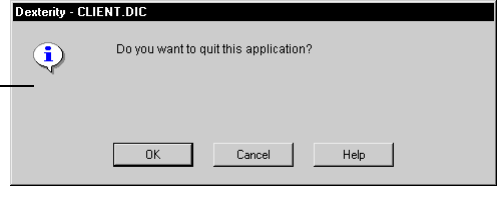
*The **ask()** function in the preceding example would display this dialog box.*



In the following example, only the first two buttons in the ask dialog box are used. Again, a help context ID is included.

```
if ask("Do you want to quit this application?", "OK", "Cancel", "",
➥ billion + 1187) = ASKBUTTON1 then
    close application;
end if;
```

*The **ask()** function in the preceding example would display this dialog box.*



assert

Description

The **assert** statement provides a mechanism for verifying the state of computation at a given point in a script.

Syntax

assert *boolean_expression* {, *message*}

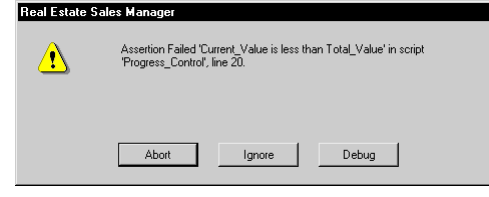
Parameters

- *boolean_expression* – An expression that you claim to be true when the **assert** statement is executed.
- *message* – An optional string containing a message to include in the dialog displayed when the assertion fails.

Comments

An *assertion* is a claim about the state of computation at the time the assertion is evaluated. Assertions are always written as statements the programmer believes to be true. You can make your code self-testing by adding assertions. An assertion that fails indicates a potential problem in the script.

When an **assert** statement is encountered, its boolean expression (the assertion) is evaluated. If the expression is true, the script will continue. If the expression is false, a dialog box like the one shown in the following illustration will be displayed.



You can choose to abort the script, ignore the assertion or start the script debugger and examine the script where the assertion failed.

The **assert** statement is a development and testing tool. The dialog box indicating an assertion failed should not be displayed to the users of your application. By default, the runtime engine will not display the dialog box, even though an assertion failed. To allow testing with the runtime engine, add the **RuntimeAsserts=TRUE** setting to the defaults file. This forces the runtime engine to display the dialog box for any assertions that fail.

Since assertion dialogs should not be seen by users, and some overhead is involved in evaluating them, we recommend that you don't compile **assert** statements for the final version of your product. To prevent **assert** statements from being compiled, don't compile your application with debug information.

Examples

The following example shows the beginning of the Progress_Control procedure from the Real Estate Sales Manager sample application. An assertion was added immediately after the procedure parameters to check the input values. The assertion states the Current_Value should never exceed the Total_Value. If this assertion fails, the script that called the procedure probably contains an error.

```
in long Current_Value;
in long Total_Value;

assert Current_Value <= Total_Value, "Current_Value is greater than
➥ Total_Value";
```

assign

Description

The **assign** statement establishes a reference to a specific resource in the application.

Syntax

assign *reference_field* **as reference to** *resource*

Parameters

- *reference_field* – A field with the reference control type to which the reference will be assigned.
- *resource* – The resource to which a reference is being established.

Comments

A reference can be stored in a global field or local field that uses the reference control type. References can also be stored in local variables or passed as parameters having the type **reference**.

If you use a reference field as a hidden field on a window, be sure the SavedOnRestart property is set to true. If it is not set to true and the form or window is restarted, the reference field will be cleared and the reference it contains will be lost.

Examples

The following example creates a reference to the Seller_Data table and assigns it to the hidden reference field, Main Table.

```
assign 'Main Table' of window Sellers as reference to table
➥ Seller_Data;
```

The following example creates a reference to the Seller ID field in the Sellers window and assigns it to the hidden reference field, Control Field.

```
assign 'Control Field' of window Sellers as reference to field
➥ 'Seller ID' of window Sellers;
```

The following example shows how to use the reference field created in the previous example. It sets the value of the field indicated by the reference.

```
field('Control Field') = "1001";
```

Related items

Additional information

Chapter 25, "References," in Volume 2 of the Dexterity Programmer's Guide

assign as key

SQL

Description

The **assign as key** statement dynamically creates a new key for a table. The parameters for the new key are used to specify the sorting order as the table's data is returned from SQL Server™.

Syntax

assign *key_variable* **as key** for table *table_name* {with *key_options*} using *table_field* {with *segment_options*} {, *table_field* {with *segment_options*}}...

Parameters

- *key_variable* – An integer variable to which the new key will be assigned. This variable will be used when the new key is applied.
- table *table_name* – The table for which the new key is being created.
- with *key_options* – An optional clause that specifies options for the key. The following table lists the options that are available.

Key option	Description
KEY_OPTION_ALLOW_DUPLICATES	Indicates that duplicate key values are expected based on the key definition.
KEY_OPTION_SQL_UNIQUE	Indicates that key values are expected to be unique based on the key definition. This is the default if no key option is specified.

It's important to correctly specify whether the new key should allow duplicates. If the new key specifies that the key values will be unique, but encounters duplicate key values, the order of the rows returned will be inconsistent.

- using *table_field* – A reference to the first field in the table to use for the new key.
- with *segment_options* – An optional clause that specifies the options to use for the segment created from the *table_field*. The following table lists the options that are available.

Key segment option	Description
KEY_SEGMENT_OPTION_DESCENDING	Segment is sorted in descending order. If this option is not specifed, the segment is sorted in ascending order.

- *table_field* with *segment_options* – Additional optional clauses that define additional segments for the key. Each additional clause should be separated by a comma.

Comments

The **assign as key** statement is typically used when retrieving data from a SQL table, but there isn't a key available that will sort the data in the desired order.

Examples

The following example creates a new key for the House_Data table, based on the City field in the table. There can be duplicate City values, so the option to allow duplicates is used.

```
local integer virtual_key;

assign virtual_key as key for table House_Data with
➥ KEY_OPTION_ALLOW_DUPLICATES using City of table House_Data;
```

The following example creates a new key for the House_Data table, based on the State and City fields in the table.

```
local integer virtual_key;

assign virtual_key as key for table House_Data with
➥ KEY_OPTION_ALLOW_DUPLICATES using State of table House_Data, City
➥ of table House_Data;
```

The following example creates a new key for the House_Data table, based on the Asking Price field in the table. The key segment will be sorted in descending order.

```
local integer virtual_key;

assign virtual_key as key for table House_Data with
➥ KEY_OPTION_ALLOW_DUPLICATES using 'Asking Price' of table
➥ House_Data with KEY_SEGMENT_OPTION_DESCENDING;
```

The following example creates a new key for the House_Data table, based on the City and Asking Price fields in the table. The Asking Price is sorted in descending order. A range is set on the City to include records for "Fargo". Notice how **clear** and **fill** are used since the Asking Price segment is sorted in descending order.

```
local integer virtual_key;

assign virtual_key as key for table House_Data with
➥ KEY_OPTION_ALLOW_DUPLICATES using 'City' of table House_Data,
➥ 'Asking Price' of table House_Data with
➥ KEY_SEGMENT_OPTION_DESCENDING;

range clear table House_Data;
clear table House_Data;

{Set the start of the range}
'City' of table House_Data = "Fargo";
fill 'Asking Price' of table House_Data;
range start table House_Data by number virtual_key;

{Set the end of the range}
clear 'Asking Price' of table House_Data;
range end table House_Data by number virtual_key;

get first table House_Data by number virtual_key;

while err() <> EOF do
    warning "House ID:" + 'House ID' of table House_Data + "   Price:"
    ➥ + str('Asking Price' of table House_Data);
    get next table House_Data by number virtual_key;
end while;
```

Related items

Commands
range where

beep

Description The **beep** statement generates sounds used to attract the user's attention.

Syntax **beep** *sound_type*

Parameters

- *sound_type* – The constant specifying the sound to use. Use one of the following constants:

  ERRORSOUND
  WARNSOUND
  INFOSOUND

Comments For Windows, the sounds that correspond to the three sound constants are set up in the Sounds control panel. The following table lists the sound constants and the corresponding event in the Sounds control panel.

Constant	Windows event
ERRORSOUND	Default sound
WARNSOUND	Question
INFOSOUND	Exclamation

If your computer doesn't have a sound card, you will get the default beep, regardless of the sound constant used.

Examples The following example uses the **beep** statement and the **warning** statement to display a warning dialog box. The boolean value Item Found indicates whether the item was found.

```
if not 'Item Found' then
    beep WARNSOUND;
    warning "The item was not found.";
end if;
```

begingroup

Description

The **begingroup** statement indicates the beginning of a group of procedures that will be processed as a single entity. This statement allows you to define the name of the group and several attributes that will be used to place this group in the appropriate process queue, either locally or at a process server.

Syntax

begingroup *group_name* {**with load factor** *factor*} {**using service** *service_name*} {**with priority** *value*} {**queue at** *time* **on** *date*} {**recur every** *interval* **[minutes | hours | days | weeks | months | years]**} {**notify** *procedure*} {**groupdone** *procedure*} {**deletable** *deletable_boolean*} **assign to** *group_ID*

Parameters

- *group_name* – A string containing the name for the group being created. This name will appear in the Process Monitor window when the group is running.

- **with load factor** *factor* – An integer value used to measure the relative amount of work required to complete this process group. For example, if you have a process group that prints two simple reports, it will require less work (therefore putting less of a load on the process server) than a process group printing seven long and complex reports. Valid values for the *factor* parameter are in the range 1 to 1000, with 1000 representing the heaviest load on the process server.

- **using service** *service_name* – Specifies a predefined set of process servers that this process group could be sent to. If this parameter isn't included, the group will be processed locally. Which server it is sent to depends upon the size, priority and load factor of the current process group and the process groups in the process queue at each process server in the named service. Since this parameter deals with processes that are to be sent to a process server, it should be used for only client/server applications.

- **with priority** *value* – An integer that assigns a level of priority to the tasks this process group will perform. Valid values for the *value* parameter are in the range 1 to 100, with 1 having the highest priority.

- **queue at** *time* **on** *date* – Assigns a date and time when the process group will be sent to the appropriate process queue. There is no guarantee that the group will begin processing at the specified time, as other process groups with higher priority levels may already be in that queue. If this parameter isn't included, or the *time* and *date* values are empty, the group will begin processing immediately.

- **recur every** *interval* **[minutes | hours | days | weeks | months | years]** – A frequency with which you want this process sent to a process queue. The *interval* specified must be an integer in the range of 1 to 10000.

If an integer outside the valid range is used, the default value of 10000 will be used.

- **notify** *procedure* – A string containing the name of a procedure on the client to be called when the process group has run to completion. This allows you to notify the user that this process group's task has been completed.

- **groupdone** *procedure* – A string containing the name of a procedure to be called when the process group has run to completion. The procedure will be executed where the group was processed. For example, if the group was processed remotely on a process server, the procedure will be executed on the process server. If the group was processed locally, the procedure will be executed locally.

- **deletable** *deletable_boolean* – A flag specifying whether the process group can be deleted from a process queue. True indicates the group can be deleted, while false indicates it can't.

- **assign to** *group_ID* – A long integer variable to which the ID generated by Dexterity for this group is assigned. The *group_ID* uniquely identifies the process group.

Comments

This statement can be used when processing shouldn't be interrupted for a group of procedures, such as posting procedures. All procedures in a group defined by **begingroup** will be treated as a single item, and placed in the client's process queue (if the **using service** *service_name* parameter isn't used) or the Process Server process queue (if the **using service** *service_name* parameter is used and the procedures are called using **call remote** statements).

*Don't mix **call remote** and **call background** calls in the same process group. This will lead to unpredictable results.*

The process group's *group_name* will appear in the Process Monitor window, followed by the number of procedures in the process group.

As processes are added to a process queue, they are inserted into the queue based upon their priority *value*. For example, if a process group of priority 50 is added to a queue, it will be inserted into the queue before any groups of priority 51 or higher, and after any of priority 50 or lower. Active process groups always run to completion and can't be interrupted by higher-priority groups.

If you use the **notify** *procedure* or **groupdone** *procedure* parameters, the named procedure must contain three in parameters. The first parameter must be a string to accommodate the name of the process group. The second parameter must be a long integer to accommodate the group ID that is assigned by the **begingroup** statement. The third parameter also must be a long integer to accommodate the completion status of the process group.

The following constants indicate the completion status:

Constant	Description
QUEUE_STATUS_CANCELED	An active process group was canceled while it was processing.
QUEUE_STATUS_COMPLETED	The process group was completed.
QUEUE_STATUS_DELETED	A pending process group was deleted.
QUEUE_STATUS_ERROR	An unknown error occurred at the Process Server and the group couldn't be completed.
QUEUE_STATUS_SHUTDOWN	A pending, deletable process was not completed because the Process Server was shut down.

We recommend that you create process groups for all background and remote processes, regardless of whether they are single procedures or groups of procedures. Then, you can take advantage of process group features such as load balancing, prioritization, and delayed queuing. All remote processes must be grouped using the ***begingroup*** *and* ***endgroup*** *statements.*

Examples

The following example defines three procedures in the "Item Posting" process group. This group will be processed in the background and will appear in the Process Monitor window on the client workstation where the process group is run. The procedure Posting_Completed will be called when the process group has completed. The group is marked as deletable, so that the user may delete it from the process queue before it begins processing. The Group_ID variable stores the unique ID that Dexterity assigns so that it may uniquely identify this process group. Using the same Group_ID with the **endgroup** statement indicates the end point of the process group.

```
local long Group_ID;

begingroup "Item Posting" notify "Posting_Completed" deletable true
➥ assign to Group_ID;

{Call the processes that will be in this process group.}
call background Post_To_Item_Open;
call background Rebuild_Item_Open;
call background Print_Item_Posting_Reports;

{Indicate the end of the process group.}
endgroup Group_ID;
```

The following example adds two procedures to the "Item Status Reports" process group. This group will be processed remotely, at a process server workstation in the "Printing Servers" service. It will be sent to the process queue at 10 p.m. tonight, and every 24 hours thereafter.

```
local long Group_ID;
local time process_time;
local date process_date;

process_date = sysdate();
process_time = mktime(22,0,0);

begingroup "Item Status Reports" using service "Printing Servers"
➥ queue at process_time on process_date recur every 24 hours
➥ assign to Group_ID;

{Call the remote processes that will be in this process group.}
call remote IV_Print_Stock_Status;
call remote IV_Print_Backorder_Report;

{Indicate the end of the process group.}
endgroup Group_ID;
```

Related items

Commands

call, **call with name**, **endgroup**

Additional information

Chapter 12, "Process Server," in the Dexterity Stand-alone Application Guide

bitclear()

Description

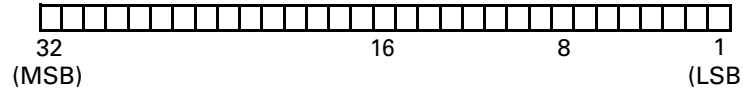
The **bitclear()** function takes a copy of the value supplied, clears the specified bit and returns the value.

Syntax

bitclear(*value*, *pos***)**

Parameters

- *value* – A tiny integer, integer or long integer value for which a bit will be cleared.
- *pos* – The position of the bit to be cleared. The value 1 indicates the least-significant bit (LSB). The position can be up to 8 for tiny integers, 16 for integers, and 32 for long integers.



Return value

The value supplied with the specified bit cleared.

Examples

The following example clears the third bit of the status variable.

```
status = bitclear(status, 3);
```

Related items

Commands

bitset(), **bittest()**

bitset()

Description

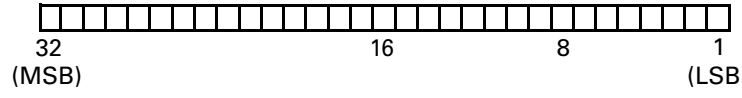
The **bitset()** function takes a copy of the value supplied, sets the specified bit and returns the value.

Syntax

bitset(*value*, *pos***)**

Parameters

- *value* – A tiny integer, integer or long integer value for which a bit will be set.
- *pos* – The position of the bit to be set. The value 1 indicates the least-significant bit (LSB). The position can be up to 8 for tiny integers, 16 for integers, and 32 for long integers.



Return value

The value supplied with the specified bit set.

Examples

The following example sets the third bit of the status variable.

```
status = bitset(status, 3);
```

Related items

Commands

bitclear(), **bittest()**

bittest()

Description

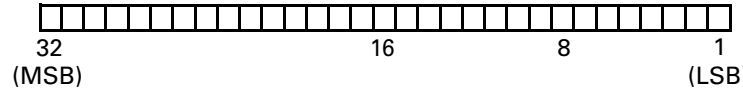
The **bittest()** function tests the status of a bit.

Syntax

bittest(*value*, *pos***)**

Parameters

- *value* – A tiny integer, integer or long integer value for which a bit will be tested.
- *pos* – The position of the bit to be tested. The value 1 indicates the least-significant bit (LSB). The position can be up to 8 for tiny integers, 16 for integers, and 32 for long integers.



Return value

A boolean. True indicates the bit was set. False indicates it was not.

Examples

The following example returns a boolean indicating whether the third bit of the status variable is set.

```
local boolean is_set;

is_set = bittest(status, 3);
```

Related items

Commands

bitclear(), **bitset()**

call

Description

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

Syntax

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

Parameters

- **background** | **foreground** | **remote** – Indicates whether the procedure will be processed in the background, foreground or remotely by a process server.

  The **remote** keyword sends the procedure to a process server in the service named by the **begingroup** statement preceding this statement. If the **begingroup** statement wasn't used, the called procedure will be run in the background.

  The **foreground** keyword starts a procedure that runs in the foreground. This can be used to assure that a block of sanScript statements is not interrupted by user interaction with the application. Refer to the examples for more information about using the **foreground** keyword.

- *procedure* – The name of the procedure you want 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 256 parameters can be passed to or returned from the called procedure.

Comments

You can use the Process Monitor to monitor the progress of procedures.

The **call** statement transfers temporary control to a procedure, passing information to it. If the **background** keyword is not used, the called procedure can pass information back to the calling script. 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. Background procedures must be completed before users are allowed to exit your application.

You can't pass a table buffer when calling a procedure to run in the background because the form where the calling script originates might be closed 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.

Examples

In the following example, a procedure named Check Customer is called to check a customer's status. The contents of the Customer Name and Sale Total fields are passed to the procedure.

```
call 'Check Customer', 'Customer Name', 'Sale Total';
```

In the following example, a form procedure named Delete Customer is attached to the Customer_Maintenance form. The Customer ID field is passed to the procedure.

```
call 'Delete Customer' of form Customer_Maintenance, 'Customer ID';
```

In the following example, the IV_Stock_Status procedure is sent to a process server in the "Printing Servers" service. Its priority level is assigned by the user's selection in the priority_level drop-down list located in the window from which this procedure is run. The three choices on this list are urgent, normal and low priority.

```
local long group_ID;
local integer importance;

{Set the priority based on the user selection.}
case priority_level
    in [1]
    importance = 1;
    in [2]
    importance = 50;
    in [3]
    importance = 99;
end case.
{Create a process group.}
begingroup Stock_group using service "Printing Servers" with
➥ priority importance assign to group_ID;
call remote IV_Stock_Status;
{End the process group.}
endgroup group_ID;
```

The following example describes an instance when you would use the **foreground** keyword when calling a procedure. The following statements open the Progress_Indicator form and set the values of the description fields.

```
open form Progress_Indicator;
'(L) Description Line1' of window Progress_Indicator of form
➥ Progress_Indicator = "Processing";
'(L) Description Line2' of window Progress_Indicator of form
➥ Progress_Indicator = "One moment please.";
```

When these statements are run as part of a background script, the Progress_Indicator form will be displayed first. Because the script is running in the background, the user could close the Progress_Control window before the remaining statements of the background script have finished. If this occurs, the two statements that reference fields in the Progress_Indicator window will generate null field address errors because the window has already been closed.

To prevent these errors, the statements to open the Progress_Indicator form and set the Description fields could be placed in a global procedure. In the background procedure, this new procedure would be called using the **foreground** keyword. All of the statements in the foreground procedure will run to completion before the user can interact with the system. After the foreground procedure has finished, the background procedure will continue.

Related items

Commands

begingroup, **call with name**, **endgroup**

Additional information

Chapter 12, "Process Server," in the Dexterity Stand-alone Application Guide

call sproc

SQL

Description

The **call sproc** statement executes the named stored procedure on the data source.

Syntax

call sproc *sproc_name, ret_val* {, *parameter, parameter, ...*}

Parameters

- *sproc_name* – A string containing the name of the stored procedure you want to run.
- *ret_val* – A long integer variable containing the error code returned by the stored procedure. By default, SQL native error codes are returned. The value 0 corresponds to no error, while the values -1 to -14 represent native SQL server error codes. You can choose to have the stored procedure return another value through the *ret_val* parameter, but this will override any native error value returned.
- *parameter* – Values, such as strings or integers, that you want to pass to or have returned from the stored procedure.

Comments

Stored procedures are precompiled SQL statements stored in a database. Using stored procedures can improve your application's performance, since all of the processing takes place at the server. Stored procedures can call other stored procedures.

The **call sproc** statement doesn't verify the existence of the named stored procedure or the items in the parameter list when the script containing it is compiled. If the procedure is unsuccessful when run in your application, the return value will be -127, the value corresponding to the SQL_SPROC_ERROR constant.

The Dexterity long integer control type corresponds to the SQL integer data type. Therefore, your stored procedure's return value must be declared as an integer, but the Dexterity ret_val *to which it is returned must be a long integer. Stored procedures must not generate a result set. Only returned values and parameters can be stored procedure outputs.*

Refer to the Chapter 45, "Stored Procedures," in Volume 1 of the Dexterity Programmer's Guide for more information about using stored procedures.

The Dexterity dictionary must contain a prototype procedure for the named stored procedure. This prototype procedure must have the same name as the stored procedure. The first non-comment line of the prototype procedure indicates what the stored procedure will return. It has the following format:

sproc returns long *variable;*

Then any additional in, out or inout parameters for the stored procedure are declared. This prototype reserves the necessary space in the client's memory for any out parameters. The **call sproc** statement must be used in the prototype procedure to make the actual call to the SQL stored procedure. When you call the prototype procedure, the SQLScriptPath procedure will be called implicitly, which adds path information to the stored procedure named in the **call sproc** statement.

Examples

The following examples show common ways to use stored procedures:

- Calling a stored procedure that doesn't require in parameters and returns only the return value.
- Calling a stored procedure that requires in parameters and returns out parameters, as well as the return value.

This stored procedure, named Count_Seniors, queries the EMPLOYEES table to find the number of employees over age 65. Since the procedure is simple and returns only one parameter, the value @Count, this procedure uses the return value as an out parameter. This method is used infrequently because the return value will override any native error code that would be returned.

```
CREATE PROC Count_Seniors
    AS
       DECLARE @Count int
       /* Count the number of employees over age 65 */
       SELECT @Count = count(first_name) from EMPLOYEES where Age > 65
       RETURN @Count
```

The following Dexterity prototype procedure, also named Count_Seniors, calls the Count_Seniors stored procedure.

```
sproc returns long Ret_Code;

call sproc "Count_Seniors", Ret_Code;
```

The following script calls the Count_Seniors prototype procedure and sets the value of the Senior Employees field. If the stored procedure is successful, num_seniors will be set to the @Count value returned by the stored procedure. Otherwise, it will be set to -127, the value corresponding to the SQL_SPROC_ERROR constant. This potential for displaying erroneous data is why return values are rarely used as out parameters.

```
local long num_seniors;

call Count_Seniors, num_seniors;
'Senior Employees' = num_seniors;
```

The following example shows a stored procedure named Profile. It requires an in parameter, and returns five out parameters. The return value isn't manipulated in this procedure; rather, it's used for the default functionality, status checking. If the stored procedure was successful, the value of the STATUS_SUCCESS constant, 0 (zero), is returned. If it failed, a value from -1 to -14 (native SQL error codes) or the SQL_SPROC_ERROR constant, -127, is returned.

```
CREATE PROCEDURE Profile(@Cust_ID smallint, @Name char output,
    @City char output, @State char output,
    @Tot_Purch numeric output, @Cust_Award char output)
AS
     SELECT @Name, @City, @State
    FROM Customer_Table
    WHERE Cust_ID = @Cust_ID
    SELECT @Tot_Purch=Q1_Sales + Q2_Sales + Q3_Sales + Q4_Sales
    IF @Tot_Purch >= 100000.00
        @Cust_Award = 'Statue'
    ELSE
    IF @Tot_Purch between 50000.00 and 99999.99
        @Cust_Award = 'Plaque'
    ELSE
        @Cust_Award = 'None'
```

The prototype procedure, also named Profile, is defined as follows. The sproc_status parameter is the return value, and must be declared using the **sproc returns** clause. The remaining parameters must be listed in the same order as in the stored procedure.

```
sproc returns long sproc_status;
in integer Cust_ID;
inout string Name;
inout string City;
inout string State;
inout currency Tot_Purch;
inout string Cust_Award;

call sproc "Profile", sproc_status, Cust_ID, Name, City, State,
➥ Tot_Purch, Cust_Award;
```

The following is a portion of a script that uses the **call** statement to run the Profile procedure.

```
local long sproc_status;

Cust_ID = Cust_ID of window Customer_Info;
call Profile, sproc_status, Cust_ID, Name, City, State, Tot_Purch,
➥ Cust_Award;
```

Related items

Additional information

Chapter 45, "Stored Procedures," in Volume 1 of the Dexterity Programmer's Guide

call with name

Description

The **call with name** statement transfers temporary control to the named procedure. The procedure to run is specified at runtime.

Syntax

call { **background** | **remote** } **with name** *procedure_name* {**in dictionary** *product_ID*} {, *parameter, parameter, ...*}

Parameters

- **background** | **remote** – Indicates whether the procedure will be processed in the background or remotely by a process server. The **background** parameter runs the procedure in the background while the computer continues running the calling script in the foreground.

- The **remote** keyword sends the procedure to a process server in the service named by the **begingroup** statement preceding this statement. If the **begingroup** statement wasn't used, the called procedure will be run in the background.

- *procedure_name* – A string expression or variable containing the name of the procedure you want to call.

If you are calling a form-level procedure, use single quotes around the procedure name, within *the string expression. Calling a form-level procedure is shown in the second example.*

- **in dictionary** *product_ID* – A clause containing the integer product ID for a third-party dictionary containing the procedure you want to call.

- *parameter* – Values, such as tables or fields you wish to pass to or return from the procedure. Up to 256 parameters can be passed to or returned from the called procedure.

Comments

You can use the Process Monitor to monitor the progress of procedures.

The **call with name** statement transfers temporary control to a procedure, passing information to it. If the **background** keyword is not used, the called procedure can pass information back to the calling script. 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. Background procedures must be completed before users are allowed to exit your application.

You can't pass a table buffer when calling a procedure to run in the background because the form where the calling script originates might be closed 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.

When using **call with name** to call procedures with optional parameters, you must include all of parameters in the call, including any optional parameters at the end of the parameter list.

Examples

In the following example, Background Procedures is a combo box listing the procedures for an application that can be processed in the background. The script will run the procedure currently selected in the field.

```
call background with name 'Background Procedures';
```

In the following example, Calculate Commissions is a form-level procedure that can be processed remotely. The script will send the procedure to a process server in the Sales_Servers service.

```
local long group_ID;

begingroup Procedure_Group using service Sales_Servers assign to
➥ group_ID;
call remote with name "'Calculate Commissions' of form 'Sales Data'";
endgroup group_ID;
```

Related items

Commands

begingroup, **call**, **endgroup**

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.

Examples

The following example uses the **case...end case** statement to set the rebate amount to be paid to customers based upon their total purchases for the current fiscal year.

```
case 'FY Total Purchases'
    in [-99999 to 999]
    {The customer purchased less than $1,000 worth of goods. They
    aren't eligible for a rebate.}
    'Rebate Award Amount' = 0;
    in [1000 to 2999]
    {The customer purchased between $1,000 and $2,999 worth of goods.
    They qualify for a $15 rebate.}
    'Rebate Award Amount' = 15;
```

```
    in [3000 to 4999]
    {The customer purchased between $3,000 and $4,999 worth of goods.
    They qualify for a $25 rebate.}
    'Rebate Award Amount' = 25;
else
    {The customer purchased over $5,000 worth of goods. They
    qualify for a $50 rebate.}
    'Rebate Award Amount' = 50;
end case;
```

The following example uses the **Login_LogIntoDataSource()** function in a change script attached to the OK button of a login window. The user selects a data source from a drop-down list named Data_Source_DDL and enters a user ID and a password.

```
local string data_source_name;
local integer status;

data_source_name = itemname(Data_Source_DDL, Data_Source_DDL);
User_ID of table User_Verify = User_ID of window Login_Window;
Password of table User_Verify = Password of window Login_Window;
status = Login_LogIntoDataSource(data_source_name, User_ID,
➥ Password);
if status <> STATUS_SUCCESS then
case status
    in [STATUS_ERROR]
        {Display a message.}
        warning "An error has occurred.";
    in [STATUS_ALREADY_LOGGED_IN]
        warning "You're already logged in.";
    in [STATUS_LOGIN_FAILED]
        warning "Login attempt failed.";
    in [STATUS_TOO_MANY_CONNECTIONS]
        warning "No connections are available.";
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* – The name of the table containing the record to read.

- **by** *key_name* | **by number** *expr* – Identifies the key to use 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.

*The **lock** option shouldn't be used unless the Allow Active Locking option has been marked for the specified table in the Table Definition window. If the Allow Active Locking option hasn't been marked for a given table, any **change** statements specifying that table and including the **lock** option will not compile.*

If your application will use the SQL database type, we recommend that you rely on passive locking instead of active locking. Using active locking with the SQL database type can cause performance degradation.

- **wait** – Indicates that the statement will wait until it can lock the record. The **wait** keyword can be used only in conjunction with the **lock** keyword. We recommend caution when using the **wait** feature. It suspends the user's computer until the record becomes available.

*The **wait** feature is not supported by SQL or Pervasive.SQL. The **wait** keyword will be ignored by Dexterity applications using these database types.*

- **refresh** – This keyword only applies when the SQL database type is used. It indicates that the cursor and client-side buffer will be refreshed, so that they'll contain current data from the table before the change occurs. Even if the change attempt fails, the refresh will occur.

*If the Pervasive.SQL or c-tree Plus database types are used, the **refresh** keyword will be ignored. It will not affect your application in any way.*

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.

The **change** statement fills only the table buffer. To transfer information to the window so the information can be displayed, use a **copy from table** statement. To transfer a single field of information from the table buffer to the window, use the **set** statement.

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

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

Examples

The following example sets the table buffer of the Customer_Master table to the Customer_ID value entered in the Customer_Entry window. The **change** statement then uses the value in the table buffer to retrieve a record from the Customer_Master table and place it in the table buffer. If the retrieval was successful, the record is then copied to the window.

```
{Set the key value of the Customer_Master table.}
'Customer ID' of table Customer_Master = 'Customer ID' of window
➥ Customer_Entry;
{Attempt to read the record.}
change table Customer_Master;
if err() = OKAY then
    {Retrieved the record. Copy it to the window.}
    copy from table Customer_Master;
else
    {Couldn't retrieve the record. Display a warning.}
    warning "This customer could not be located.";
    restart field;
end if;
```

The following example uses the **change** statement to retrieve each record in the Customer_Master table. The **by number** *expr* parameter is used to indicate that the second key created for the table will be used to locate records. After each record is read, the Contact field is set to Steve and the record is saved.

```
{Read the first record in the table.}
change first table Customer_Master by number 2;
while err() = OKAY do
    {Retrieved a record. Change the contact.}
    Contact of table Customer_Master = "Steve";
    {Save the changed record.}
    save table Customer_Master;
    {Read the next record.}
    change next table Customer_Master;
end while;
```

Related items

Commands

check error, **edit table**, **err()**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

change item

Description

The **change item** statement changes the text for an item in a menu or a list field. List fields include list boxes, multi-select list boxes, combo boxes, drop-down lists, button drop lists and visual switches.

Syntax

change item *number* **of** [**field** / **menu**] *name* to *string_expression*

Parameters

- *number* – An integer indicating the position of the item in the menu or field for which the text will be changed.
- *name* – The name of the field or menu for which an item is being changed.
- *string_expression* – The string you want to use for the item.

Comments

Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, should not be counted when determining the position of a menu item for use in the *number* parameter.

Examples

The following example changes the text of the second item of the Status list field.

```
change item 2 of field 'Status' to "Discontinued";
```

The following example changes the first item of the Markup menu.

```
change item 1 of menu 'Markup' to "25%";
```

Related items

Commands

add item, **delete item**, **finddata()**, **finditem()**, **itemname()**, **redraw**, **Field_GetInsertPosFromVisualPos()**, **Field_GetVisualPosFromInsertPos()**

changed()

Description

The **changed()** function returns a boolean value indicating whether any of the non-button fields in a window or form have been changed by the user since the window or form was opened or the **clear changes** statement was run.

Syntax

changed ([**form** | **window**] *name***)**

Parameters

- **form** | **window** – The type of resource you're checking.
- *name* – The name of the form or window to be checked.

Return value

Boolean

Comments

A typical use for this function is to ascertain whether the user is closing a form or window in which changes have been made, without having saved the changes.

You can also use the **changed()** function with a scrolling window to find out whether the line change script for the scrolling window is going to run.

Examples

The following example uses the **changed()** function to ascertain whether the Customer_Master form has changed. If changes to the form have been made, the **save table** statement is used to save the changes.

```
if changed(form Customer_Master) then
    save table Customers;
end if;
```

Related items

Commands

clear changes, **force changes**

char()

Description

The **char()** function returns the ASCII character corresponding to an integer value.

Syntax

char(*integer***)**

Parameters

- *integer* – An integer value in the range of 0 to 255, for which you want the ASCII character equivalent. If *integer* is set to a value out of the range 0 to 255, a null string will be returned.

Return value

String

Examples

The following example sets the string field, single_letter, to the value 'A', which corresponds to the ASCII value 65.

```
single_letter = char(65);
```

This example sets the string field, single_letter, to the character value of an integer field, user_int.

```
single_letter = char(user_int);
```

Related items

Commands

ascii()

Additional information

Appendix C, "ASCII Character Codes," in Volume 2 of the Dexterity Programmer's Guide

check command

Description

The **check command** statement sets a command to the checked state. The command will appear marked in any menu or toolbar in which it is displayed.

Syntax

check command *command_name* {, *command_name*}

Parameters

- *command_name* – The name of the command to be checked.

Comments

Use the **Command_GetBooleanProperty()** function to find out whether a command is checked or unchecked.

Examples

The following example checks the CMD_ComputeTaxes command. The command's appearance will be changed any place the command is displayed, such as a menu or toolbar.

```
check command CMD_ComputeTaxes;
```

Related items

Commands

uncheck command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

check error

Description

The **check error** statement checks the result of the last operation on the specified table, and displays a corresponding alert 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 wish the last table operation checked.

Comments

The operation that caused the error and the table accessed will be displayed, as well as the alert message. For example, if the **get** statement was used to retrieve a non-existent record from the Seller_Data table, the **check error** statement would display the following alert message:

Examples

In the following example, a record is being retrieved from the Customer_Master table. The **check error** statement checks whether any errors occur as the information is being retrieved and displays an appropriate alert message if an error occur.

```
'Customer Number' of table Customer_Master = 'Customer Number' of
➥ window Customer_Maintenance;
get table Customer_Master;
{Display any message describing any error.}
check error;
```

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

```
'Customer Number' of table Customer_Info = 'Customer Number' of
➥ window Customer_Maintenance;
'Customer Number' of table Customer_Master = 'Customer Number' of
➥ window Customer_Maintenance;
get table Customer_Master;
get table Customer_Info;
{Display any message describing any error.}
check error table Customer_Master;
check error table Customer_Info;
```

Related items

Commands

err(), **naterr()**

check field

Description

The **check field** statement selects an item in a multi-select list box or places a check mark next to an item in a button drop list.

Syntax

check field *field_name*, *number*

Parameters

- *field_name* – The name of the field containing the item you want selected or marked.
- *number* – An integer indicating which item to select or mark.

Comments

Use the **finditem()** function to retrieve the number of an item from a multi-select list box or any other list field. The retrieved number can then be used as the *number* parameter.

Always use the **check field** *statement to set the values in a multi-select list box. Never use the* ***set*** *statement to change the items selected in a multi-select list box.*

Examples

The following example selects the first item in the Payment Methods multi-select list box.

```
check field 'Payment Methods', 1;
```

The following example is the change script for a button drop list. It uses the **checkedfield()** function to find out whether the selected item was checked. If it was, it removes the check mark next to the item that was selected. Otherwise, it checks the item.

```
if checkedfield('Reports', 'Reports') then
    uncheck field 'Reports', 'Reports';
else
    check field 'Reports', 'Reports';
end if;
```

Related items

Commands

checkedfield(), **finditem()**, **uncheck field**

check menu

Description

The **check menu** statement places a check mark next to a menu item.

Syntax

check menu *menu_name*, *number*

Parameters

- *menu_name* – The name of the menu containing the item you want checked.
- *number* – An integer representing the placement of the item to be checked in the menu.

Comments

The **check menu** statement cannot be used to place a check mark next to grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste. For more information about menu items that can be checked and unchecked, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

Examples

The following example places a check mark next to the third item in the Transactions menu.

```
check menu Transactions, 3;
```

Related items

Commands
checkedmenu(), **uncheck menu**

checkedfield()

Description

The **checkedfield()** function returns a boolean value indicating whether a specified position in a multi-select list box is selected or an item in a button drop list is checked.

Syntax

checkedfield(*field, position***)**

Parameters

- *field* – The name of the multi-select list box or button drop list.
- *position* – An integer indicating the position of an item in the multi-select list box or button drop list.

Return value

A boolean. The value true indicates the item was marked, while false indiciates it was not.

Examples

The following example ascertains whether the first item in a multi-select list box named Taxes is selected. If it is, the value of the State Tax field will be added to the value of the Total Tax field.

```
if checkedfield(Taxes, 1) then
    'Total Tax' = 'Total Tax' + 'State Tax';
end if;
```

The following example is the change script for a button drop list. It uses the **checkedfield()** function to find out whether the selected item was checked. If it was, it removes the check mark next to the item that was selected. Otherwise, it checks the item.

```
if checkedfield('Reports', 'Reports') then
    uncheck field 'Reports', 'Reports';
else
    check field 'Reports', 'Reports';
end if;
```

Related items

Commands

check field, **uncheck field**

checkedmenu()

Description

The **checkedmenu()** function returns a boolean value indicating whether a specified position in a menu has a check mark appearing next to it.

Syntax

checkedmenu(menu *menu_name*, *position***)**

Parameters

- **menu** *menu_name* – The name of the menu.
- *position* – Integer indicating the position of a menu item in the menu.

Return value

Boolean

Comments

The **checkedmenu()** function cannot be used with grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste. For more information about menu items that the **checkedmenu()** function can be used with, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

Examples

The following example checks a menu named Report Options to ascertain whether the second menu item has a check appearing next to it. If it does, then a report named Customer List will be printed.

```
if checkedmenu (menu 'Report Options', 2) then
    run report 'Customer List';
end if;
```

Related items

Commands

check menu, **uncheck menu**

clear changes

Description

The **clear changes** statement clears the change flag for a window or a form. This flag is tested by the **changed()** function.

Syntax

clear changes [**form** | **window**] *name*

Parameters

- **form** | **window** – Indicates the type of the object for which you want to clear the change flag.
- *name* – The name of the form or window for which you want to clear the change flag.

Comments

You can also use this statement with a scrolling window to clear the change flag that causes the scrolling window line change script to run.

Examples

The following example clears the change flag for the form named Invoice.

```
clear changes form Invoice;
```

The following example clears the change flag for the Customer_List_Scroll scrolling window, preventing the scrolling window's line change script from running;

```
clear changes window Customer_List_Scroll;
```

Related items

Commands

changed(), **force changes**

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. To clear multiple fields, list the name of each field separated by a comma.

Comments

Clearing a field removes the data currently in the field. Because fields to be cleared can be located in different tables or windows, be sure to fully qualify the fields as needed.

When a list field (list box, multi-select list box, combo box, drop-down list, button drop list or visual switch) is cleared, all of the static values currently displayed in the field are removed. This includes the values defined for the list's data type. To redisplay these values, close the window, then reopen it. This will not redisplay items added to the list using the **add item** statement, unless those items were added in the window pre script, which will run when the window is reopened.

When a composite field is cleared, the individual components of the composite are cleared.

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

Examples

The following example clears multiple fields in different windows.

```
clear field 'Payment Type' of window Invoice, 'Payment Method' of
➥ window Invoice, 'Payment Date' of window Payment_Entry;
```

clear force change

Description

The **clear force change** statement stops a field's change script from automatically running once the **force change** statement has been issued for that field.

Syntax

clear force change *field_name*

Parameters

- *field_name* – The name of the field whose change script is forced to run using the **force change** statement.

Examples

The following example removes the force change applied to a Beginning Balance field.

```
if empty ('Beginning Balance') then
     clear force change 'Beginning Balance';
     warning "Please enter a beginning balance.";
end if;
```

Related items

Commands

force changes

clear form

Description

The **clear form** statement clears a specified form.

Syntax

clear form *form_name*

Parameters

- *form_name* – The name of the form to be cleared.

Comments

Clearing a form removes the data currently in each field in each of the form's windows.

When a form is cleared, the change flag for that form is cleared as well. Consequently, the **changed()** function will return false immediately after a form has been cleared.

Examples

The following example clears the form Customer_Master.

```
clear form Customer_Master;
```

Related items

Commands

changed()

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 will be cleared.

Comments

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

Examples

The following example clears the table buffer for the Invoice_Header table.

```
clear table Invoice_Header;
```

clear window

Description

The **clear window** statement clears a window or scrolling window.

Syntax

clear window *window_name*

Parameters

- *window_name* – The name of the window to be cleared.

Comments

Clearing a window removes the data currently in each field in the window. Clearing a scrolling window removes all data currently in the scrolling window, and removes the add-line and all fields in the scrolling window as well.

When a window is cleared, the change flag for that window is cleared as well. Consequently, the **changed()** function will return false immediately after a window has been cleared.

Examples

The following example clears the fields in the Invoice window.

```
clear window Invoice;
```

The following example clears the Inventory_Lookup scrolling window.

```
clear window Inventory_Lookup;
```

Related items

Commands
changed()

close application

Description

The **close application** statement closes the current application in the same way choosing the Exit menu item closes an application.

Syntax

close application

Parameters

- None

Comments

The **close application** statement closes the current application from within a script. The user can also close the application dictionary by choosing the Exit menu item.

The window and form post scripts for any open windows will be run. These scripts allow the user the opportunity to stop the close process. If any background tasks are running, the close process will be stopped.

Examples

In the following example, the script ascertains whether the User Name field has a value. If it doesn't, the application is closed.

```
if empty ('User Name') then
    warning "The user name wasn't specified. Quitting application.";
    close application;
end if;
```

close form

Description The **close form** statement closes a specified form.

Syntax **close form** *form_name*

Parameters

- *form_name* – The name of the form to be closed;

Comments Closing a form causes the form post script to run. It also closes all open windows on that form. Each window's post script will run.

*Use caution when using the **close form** statement. If you use this statement to close the form that contains the script that is running, any statements following the **close form** statement will not run.*

Examples The following example closes the Inventory_Maintenance form.

```
close form Inventory_Maintenance;
```

Related items

Commands

open form, **open form with name**

close palettes

Description

The **close palettes** statement closes all palettes currently open.

Syntax

close palettes

Parameters

- None

Comments

This command doesn't close any windows in the form from which the command was run. If the **abort close** statement is run for any window, no further palettes will be closed and the script that ran the **close palettes** statement will stop.

Examples

The following example closes all open palettes.

```
close palettes;
```

Related items

Commands

abort close, **close windows**

close table

Description

The **close table** statement closes a specified table.

Syntax

close table *table_name*

Parameters

- *table_name* – The name of the table to be closed.

Comments

Closing tables with large table buffers can make a form with many tables operate more efficiently. Tables are automatically opened when data is retrieved from them. Under ordinary conditions you won't have to close any tables.

Examples

The following example closes the Inventory_Data table.

```
close table Inventory_Data;
```

Related items

Commands

open table

close window

Description The **close window** statement closes a specified window.

Syntax **close window** *window_name*

Parameters

- *window_name* – The name of the window to be closed.

Comments Closing a window causes the window post script to run.

Examples The following example closes the Customer Maintenance window.

```
close window 'Customer Maintenance';
```

Related items

Commands
open window

close windows

Description

The **close windows** statement closes all standard windows and palettes.

Syntax

close windows

Parameters

- None

Comments

This command doesn't close tool bar windows or any windows in the form from which the command was run. If the **abort close** statement is run for any window, no further windows will be closed and the script that ran the **close windows** statement will stop.

Examples

The following example closes all standard windows and palettes.

```
close windows;
```

Related items

Commands

abort close, **close palettes**

column()

Description

The **column()** function retrieves the value from a specific field in a table.

Syntax

column(*name***) of table** *table_name*

Parameters

- *name* – A string specifying the field to retrieve.
- **of table** *table_name* – The table from which to retrieve the value.

Return value

The value from the specified field. The data type will be the same as the data type of the field being retrieved.

Comments

The **column()** function is typically used in the trigger processing procedure for cross-dictionary database triggers. It is used to retrieve field values from the anonymous table passed into the trigger processing procedure.

Examples

The following example is the trigger processing procedure that runs when a record is deleted from the Applicants table. It deletes the Internet information record corresponding to the applicant record that was deleted. The **column()** function retrieves the value in the Applicant Number field from the anonymous table passed into the trigger processing procedure.

```
{Name: Delete_Applicant_Internet_Info}
inout anonymous table 'Triggered Table';

release table Appl_Internet_Info;
'Applicant Number' of table Appl_Internet_Info =
➥ column("Applicant Number") of table 'Triggered Table';
change table Appl_Internet_Info;
if err() = OKAY then
    remove table Appl_Internet_Info;
end if;
```

Related items

Additional information
Chapter 13, "Cross-dictionary triggers," in the Integration Guide

continue

Description

The **continue** statement allows you to continue the current instance of the specified type of loop.

Syntax

continue [**for** | **repeat** | **while**]

Parameters

- **for** | **repeat** | **while** – A keyword indicating the type of loop to be continued.

Comments

This statement is useful for managing navigation within nested loops.

Examples

The following example checks the Balance field of the Customer_Balances table for customers who have dropped below the minimum balance.

```
local long count, i;

count = countrecords(table Customer_Balances);
for i = 1 to count do
    get next table Customer_Balances;
    if Balance of table Customer_Balances >= 300 then
        continue for;
    end if.
    {The customer's balance is below the minimum. Call the Add_Fee
     procedure to charge the customer a service fee.}
    call background Add_Fee, Customer_ID of table Customer_Balances,
    ➥ Balance of table Customer_Balances;
end for;
```

Related items

Commands

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

copy from field to field

Description

The **copy from field to field** statement copies matching components from one composite to another. Non-matching components are not affected.

Syntax

copy from field *source_field* **to field** *destination_field*

Parameters

- *source_field* – The name of the composite you want to copy components from.
- *destination_field* – The name of the composite you want to copy components to.

Comments

Components are matched based on their field IDs. This means a specific component in the source composite and a specific component in the destination composite must be based on the same global field resource in order to match.

Examples

The following example copies the matching components from the Note_Obj composite to the Window_Status composite.

```
copy from field Note_Obj to field Window_Status;
```

copy from table

Description

The **copy from table** statement transfers the contents of all auto-copy fields from the specified table buffer to the windows for the current form.

Syntax

copy from table *table_name*

Parameters

- *table_name* – The name of the table buffer you wish to copy field contents from.

Comments

Window fields are specified as auto-copy using the AutoCopy property in the Properties window.

Examples

In the following example, a record is read from the Customer_Master table. If the record previously existed in the table, the **copy from table** statement copies the information from the table buffer of the Customer_Master table to the windows for the form.

```
'Customer ID' of table Customer_Master = 'Customer ID' of window
➥ Customer_Maintenance;
edit table Customer_Master;
if editexisting() then
    {Copy table buffer to window fields.}
    copy from table Customer_Master;
end if;
```

Related items

Commands

copy from table to window, **copy from window to table**, **copy to table**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

copy from table to table

Description

The **copy from table to table** statement copies the contents of matching fields from one table buffer to another. Non-matching fields aren't affected.

Syntax

copy from table *source_table* **to table** *destination_table*

Parameters

- *source_table* – The name of the table buffer you want to copy the field contents from.
- *destination_table* – The name of the table buffer you want to copy the field contents to.

Comments

Fields are matched based upon their field IDs. This means a specific field in the source table and a specific field in the destination table must use the same global field resource in order to match.

The **copy from table to table** statement is useful for copying records from one table to another. A procedure using anonymous tables can be set up to allow transferring records between tables that will be specified when your application is running.

Examples

The following procedure transfers all of the Customer Lead records from the Lead_List table to the Customer_List table. The two tables use the same ID Number, Name, Address, City, State, ZIP Code and Phone Number fields, so the **copy from table to table** statement can be used to transfer the information.

```
get first table Lead_List;
while err() = OKAY do
    copy from table Lead_List to table Customer_List;
    save table Customer_List;
    get next table Lead_List;
end while;
```

copy from table to window

Description

The **copy from table to window** statement transfers the contents of all auto-copy fields from the specified table buffer to the specified window for the current form.

Syntax

copy from table *table_name* **to window** *window_name*

Parameters

- *table_name* – The name of the table buffer you want to copy the field contents from.
- *window_name* – The name of the window you want to copy the field contents to.

Comments

The **copy from table to window** statement can access table buffers and windows for only the current form. The **copy from table to window** statement *can't* be used to copy information from a table buffer in one form to a window in another form.

You *can't* use the **copy from table to window** statement in a procedure because the current form isn't known and can't be specified. You can use the statement in a form level procedure because the current form is implied by where the form level procedure is attached.

Window fields are specified as auto-copy using the AutoCopy property in the Properties window.

Examples

This example reads a record from the Customer_Master table. If the record exists, the information is copied from the Customer_Master table buffer to the Customer_Maintenance window for the current form. Other windows for the form are unaffected.

```
'Customer ID' of table Customer_Master = 'Customer ID' of window
➥ Customer_Maintenance;
edit table Customer_Master;
if editexisting() then
    {Copy table buffer to window fields.}
    copy from table Customer_Master to window Customer_Maintenance;
end if;
```

Related items

Commands

copy from table, **copy from window to table**, **copy to table**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

copy from window to table

Description

The **copy from window to table** statement transfers the contents of all auto-copy fields from the specified window to the specified table buffer for the current form.

Syntax

copy from window *window_name* **to table** *table_name*

Parameters

- *window_name* – The name of the window you want to copy the field contents from.
- *table_name* – The name of the table buffer you want to copy the field contents to.

Comments

The **copy from window to table** statement can access table buffers and windows for only the current form. The **copy from window to table** statement *can't* be used to copy information from a window in one form to a table buffer in another form.

You *can't* use the **copy from window to table** statement in a procedure because the current form isn't known and can't be specified. You can use the statement in a form level procedure because the current form is implied by where the form level procedure is attached.

Window fields are specified as auto-copy using the AutoCopy property in the Properties window.

Examples

The following example copies the contents of the auto-copy fields for the Customer_Maintenance window in the current form to the table buffer for the Customer_Master table. Fields aren't copied from any other windows in the form. The table is then saved.

```
{Copy window fields to table buffer.}
copy from window Customer_Maintenance to table Customer_Master;
save table Customer_Master;
restart form;
```

Related items

Commands

copy from table, **copy from table to window**, **copy to table**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

copy from window to window

Description

The **copy from window to window** statement copies the contents of matching fields from one window to another. Non-matching fields are not affected.

Syntax

copy from window *source_window* **to window** *destination_window*

Parameters

- *source_window* – Then name of the window you want to copy the field contents from.
- *destination_window* – The name of the window you want to copy the field contents to.

Comments

Fields are matched based on their field IDs. This means a specific field in the source window and a specific field in the destination window must use the same local or global field resource in order to match.

Examples

The following example copies the fields in the Houses window to the matching fields in the House Descriptions window.

```
copy from window 'Houses' to window 'House Descriptions';
```

copy to table

Description

The **copy to table** statement transfers the contents of all auto-copy fields from the windows for the current form to the specified table buffer.

Syntax

copy to table *table_name*

Parameters

- *table_name* – The name of the table buffer you wish to copy data into.

Comments

Exercise caution when using the **copy to table** statement when the same field is used on several windows of a form. The **copy to table** statement will copy the fields from the main window to the table buffer, then copy fields from any child windows (other windows for the form) to the table buffer. If the common field on the child window has a different value, the contents of that field will be copied to the table buffer, replacing any value from the field on the main window. This will occur even if the child window isn't open. One solution is to be sure that all the fields common to several windows have the same value. Another solution is to set the table buffer to the contents of the field after the **copy to table** statement.

Window fields are specified as auto-copy using the AutoCopy property in the Properties window.

Examples

The following example copies the contents of the auto-copy fields for the windows on the current form to the table buffer for the Customer_Master table. The table is then saved.

```
{Copy data to table buffer.}
copy to table Customer_Master;
save table Customer_Master;
restart form;
```

Related items

Commands

copy from table, **copy from table to window**, **copy from window to table**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

countitems()

Description

The **countitems()** function returns the number of items in a menu or list field. List fields include list boxes, multi-select list boxes, combo boxes, drop-down lists, button drop lists and visual switches.

Syntax

countitems([**field** / **menu**] *name***)**

Parameters

- *name* – The name of the menu or list field for which you want to return the number of items in. The field must be a window field.

Return value

Integer

Comments

Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, are not counted when the number of items in a menu is returned.

Examples

The following example sets a local variable named count to the number of items in a list field named Shipping Methods.

```
local integer count;

count = countitems(field 'Shipping Methods');
```

The following example sets a local variable named count to the number of items in the Help menu, which is part of the Main Menu form.

```
local integer count;

count = countitems(menu 'Help' of form 'Main Menu');
```

Related items

Commands

add item, **delete item**, **finditem()**, **insert item**, **itemname()**

Additional information

Chapter 6, "Data Types," in Volume 1 of the Dexterity Programmer's Guide

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.

Examples

The following example sets a local variable named Customer_Count to the number of records in the Customer_Master table.

```
local long customer_count;

customer_count = countrecords(table Customer_Master);
```

currencydecimals()

Description

The **currencydecimals()** function returns the current value of the operating system's decimal digits setting.

Syntax

currencydecimals()

Parameters

- None

Return value

Integer

Examples

The following example uses the value in the Amount currency field and converts it to a string using the **format()** function. It adds a currency symbol, a thousands separator and a minus sign if a negative value is formatted. The **currencydecimals()** function retrieves the default number of decimal places used by the operating system. The retrieved value is then used to define the decimal places parameter of the **format()** function.

```
local string result;
local integer dec_places;

{Retrieve the default number of decimal places to use.}
dec_places = currencydecimals();
{The value in Amount field is displayed as 12284.5345. The value
displayed in l_result will be $12,284.53.}
result = format(Amount, true, true, dec_places, MINUSNEG);
```

Related items

Commands

format(), **round()**, **truncate()**

currentcomponent()

Description

The **currentcomponent()** function returns an integer indicating which component of a composite the focus moved from.

Syntax

currentcomponent()

Parameters

- None

Return value

Integer

Comments

The **currentcomponent()** function is used in the pre, change and post scripts for composite fields for which the pre, change and post scripts will be run for each component, as well as the composite as a whole. The Call Component Scripts option in the Data Type Definition window specifies whether scripts will run for components.

The value returned from the **currentcomponent()** function specifies which component of the composite the focus just moved from. The following table shows which scripts run and the value returned by the **currentcomponent()** function when a user moves into, through, and out of a three-component composite field.

User action	Scripts run	Value returned
Move into the first component.	Pre script (for the entire field) Pre script (for the first component)	0 1
Change the first component and tab into the second.	Change script (for the first component) Post script (for the first component) Pre script (for the second component)	1 1 2
Change the second component and tab into the third.	Change script (for the second component) Post script (for the second component) Pre script (for the third component)	2 2 3
Change the third component and move out of the composite.	Change script (for the third component) Post script (for the third component) Change script (for the entire field) Post script (for the entire field)	3 3 0 0

Component scripts are useful for validating portions of the composite rather than validating the entire composite at once.

Examples

The following example is a change script for the Part Number composite field, which was set to use component scripts. The portion of the change script corresponding to the entire field (**currentcomponent()** = 0) attempts to retrieve an existing entry. The portion of the change script corresponding to the Vendor Number component (**currentcomponent()** = 1) verifies that the vendor exists.

```
if (currentcomponent() = 0) then
    {Change script for entire Part Number field.}
    {Retrieve the information about the part.}
    'Part Number' of table Current_Inventory = 'Part Number' of
    ➥ window Inventory_Maintenance;
    change table Current_Inventory;
    if err() = OKAY then
        copy from table Current_Inventory;
    end if;
    else if (currentcomponent() = 1) then
        {Validate the Vendor component.}
        'Vendor Number' of table Vendors =
        ➥ 'Part Number:Vendor Number';
        get table Vendors;
        if err() <> OKAY then
            warning "This vendor record hasn't been entered.
            ➥ Please select another.";
            restart field;
        end if;
    end if;
end if;
```

Related items

Additional information

Chapter 9, "Composites," in Volume 2 of the Dexterity Programmer's Guide

datatype()

Description

The **datatype()** function returns the data type associated with a field. This is especially useful for anonymous fields.

Syntax

datatype({**field**} *fieldname***)**

Parameters

- *fieldname* – The name of the field whose data type you want to retrieve.

Return value

An integer corresponding to one of the following constants:

DATATYPE_BOOLEAN	DATATYPE_NON_NATIVE_LIST_BOX
DATATYPE_BUTTON_DROP_LIST	DATATYPE_PICTURE
DATATYPE_CHECK_BOX	DATATYPE_PROGRESS_INDICATOR
DATATYPE_COMBO_BOX	DATATYPE_PUSH_BUTTON
DATATYPE_COMPOSITE	DATATYPE_RADIO_BUTTON
DATATYPE_CURRENCY	DATATYPE_RADIO_GROUP
DATATYPE_VCURRENCY	DATATYPE_REAL
DATATYPE_DATE	DATATYPE_REFERENCE
DATATYPE_DROP_DOWN_LIST	DATATYPE_SCROLLING_WINDOW
DATATYPE_HORIZONTAL_LIST_BOX	DATATYPE_STRING
DATATYPE_INTEGER	DATATYPE_TEXT
DATATYPE_INVALID	DATATYPE_TIME
DATATYPE_LIST_BOX	DATATYPE_TINY_INTEGER
DATATYPE_LIST_VIEW	DATATYPE_TREE_VIEW
DATATYPE_LONG_INTEGER	DATATYPE_UNKNOWN
DATATYPE_MULTI_SELECT_LIST_BOX	DATATYPE_VISUAL_SWITCH

Examples

The following example uses the **datatype()** function to verify that the field passed as an anonymous parameter to a procedure is a list view field. If it is not, a message is displayed and processing is stopped.

```
inout anonymous field list_view_field;

{Verify that the field is a list view.}
if datatype(field list_view_field) <> DATATYPE_LIST_VIEW then
    error "You must pass a list view field to this procedure.";
    abort script;
end if;
```

day()

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

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

Parameters

- *date* – A date or datetime value.

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

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

```
local integer day_of_month;

day_of_month = day(this_date);
```

Related items

Commands

addmonth(), **dow()**, **eom()**, **mkdate()**, **month()**, **setdate()**, **sysdate()**, **year()**

debug

Description

The **debug** statement creates a dialog box that displays the specified string. The dialog box will have one button labeled OK.

Syntax

debug *string_expression*

Parameters

- *string_expression* – A string containing the message that will appear in the dialog box.

Comments

Debug messages will be displayed in test mode when Show Debug Messages under the Debug menu is marked. If this option isn't selected, the debug messages won't be displayed. Debug messages are never displayed when the application dictionary is used with the runtime engine.

Debug messages are useful in tracking the progress of a script while locating problems. They allow you to ascertain exactly which command in a script has been run when an event occurs.

Don't use the debug statement in a window activate script. The application will become suspended in an infinite loop.

Examples

In the following example, the table is ready to be saved. The debug message alerts the programmer that this is about to happen by displaying the message, "Ready to save."

```
debug "Ready to save.";
save table Customer_Master;
```

debugger stop

Description

The **debugger stop** statement opens the Script Debugger window and causes the current script to stop processing at that point, as if a breakpoint had been encountered.

Syntax

debugger stop

Parameters

- None

Comments

The **debugger stop** statement is useful for debugging code that is cumbersome to access. Because the statement is compiled directly into the code, it isn't necessary to set a breakpoint each time the script is run.

When used with the runtime engine, the **debugger stop** statement will stop only when the ScriptDebugger=TRUE setting has been added to the Dex.ini file.

Examples

In the following example, the processing will be suspended and the Script Debugger will be displayed before the save operation is performed.

```
debugger stop;
save table Customer_Master;
```

decrement

Description

The **decrement** statement allows you to decrement, or decrease, a numeric value by a specified amount.

Syntax

decrement *number* {**by** *numeric_expression*}

Parameters

- *number* – An integer, long, currency, variable currency, time or date field or variable that you want to decrement.
- *numeric_expression* – A field or variable of the same control type as the *number* parameter, by which *number* is decreased. If no *numeric_expression* is indicated, the default decrement value is 1. The following default values are supported for each numeric control type:

Control type	Default decrement value
integer	1
long	1
currency	1 currency unit
variable currency	1 currency unit
time	1 minute
date	1 day

Examples

The following example decreases an existing sequence number by five.

```
decrement 'Sequence Number' by 5;
```

In the following example, no numeric expression is indicated, so the resulting sequence number is decreased by 1.

```
decrement 'Sequence Number';
```

The following example decreases a date value by 1. If the due date is 11/25/95, the new due date will be 11/24/95.

```
decrement 'Due Date';
```

Related items

Commands

increment

default form to

Description

The **default form to** statement provides default qualification for form names within a script. It specifies the the form to use in the script, when a form is not explicitly qualified.

Syntax

default form to *form_name*

Parameters

- *form_name* – The form that will be the default form for the current global procedure or global function.

Comments

The following guidelines must be observed when using the **default form to** statement:

- It must be used only in global procedures and global functions.
- A script can have only one **default form to** statement.

This statement is often used with the **default window to** statement.

Examples

The following example is a global procedure that updates the Progress_Control form. The **default form to** statement is used so that the "of form Progress_Control" qualifier doesn't need to be included in each reference to the items in the Progress_Control form.

```
in long Current_Value;
in long Total_Value;

default form to Progress_Control;

{Don't update the Progress_Control form if it isn't open.}
if isopen(form Progress_Control) then
    {Progress_Control is open, so update it.}
    if Current_Value >= Total_Value then
        Progress_Bar of window Progress_Control = 100;
    else
        Progress_Bar of window Progress_Control =
        ➥ integer(((Current_Value*1.0/Total_Value) * 100.0));
    end if;
end if;
```

Related items

Commands
default window to

default roundmode to

Description

The **default roundmode to** statement sets the default rounding behavior for calculations performed in the current script.

Syntax

default roundmode to *mode*

Parameters

- *mode* – An integer corresponding to one of the following constants:

Constant	Description	Examples*
ROUNDMODE_UP	Always round up.	4.651 –> 4.66 4.655 –> 4.66 4.659 –> 4.66
ROUNDMODE_DOWN	Always round down.	4.651 –> 4.65 4.655 –> 4.65 4.659 –> 4.65
ROUNDMODE_HALF_UP	If the only digit following the digit to be rounded is 5, round up.	4.675 –> 4.68 4.6751 –> 4.68
ROUNDMODE_HALF_DOWN	If the only digit following the digit to be rounded is 5, round down.	4.675 –> 4.67 4.6751 –> 4.68
ROUNDMODE_HALF_EVEN	If the only digit following the digit to be rounded is 5 and the preceding digit is odd, round up. Otherwise, truncate.	4.675 –> 4.68 4.685 –> 4.68
ROUNDMODE_CEILING	Always round toward positive infinity.	4.655 –> 4.66 -4.655 –> -4.65
ROUNDMODE_FLOOR	Always round toward negative infinity.	4.655 –> 4.65 -4.655 –> -4.66

*Rounding to two decimal places

Comments

Functions such as **round()** and **truncate()** have their own rounding mode. They don't use the mode specified by the **default roundmode to** statement.

Examples

The following example computes an interest amount. The default roundmode to statement causes all calculations to round down.

```
in vcurrency amount;
in vcurrency rate;
out vcurrency interest;

default roundmode to ROUNDMODE_DOWN;
interest = amount * rate;
```

default window to

Description

The **default window to** statement provides default qualification for window names within a script. It specifies the the window to use in the script, when a window is not explicitly qualified.

Syntax

default window to *window_name*

Parameters

- *window_name* – The window that will be the default window for the current form-level or global procedure or function.

Comments

The following guidelines must be observed when using the **default window to** statement:

- It must be used only in form-level or global procedures or functions. If it is used in global procedures or functions, it must be used after the **default form to** statement.
- Each script can have only one **default window to** statement.

Examples

The following example is a form-level procedure that updates the Monthly_Payment window. The **default window to** statement is used so that the "of window Monthly_Payment" qualifier doesn't need to be included in each reference to the fields on the window.

```
default window to Monthly_Payment;

{Calculate the house price.}
clear '(L) Monthly_Payment';
if not empty('(L) DownPayment') then
    '(L) Price' = 'Asking Price' - '(L) DownPayment';
else
    clear '(L) Price';
end if;

{Calculate the closing costs.}
if empty('(L) Price') or empty('(L) Closing Cost Percentage') then
    clear '(L) Closing Costs';
else
    '(L) Closing Costs' = '(L) Price' * '(L) Closing Cost Percentage'
    ➥ / 100.0;
end if;
```

```
{Calculate the amount financed.}
if not empty('(L) Price') and not empty('(L) Closing Costs') then
    '(L) Amount Financed' = '(L) Price' + '(L) Closing Costs';
else
    clear  '(L) Amount Financed';
end if;
```

Related items

Commands

default form to

delete item

Description

The **delete item** statement deletes a string from any type of list field: a list box, multi-select list box, combo box, drop-down list, button drop list or visual switch.

Syntax

delete item *expression* **from** {**field**} *window_field*

Parameters

- *expression* – An integer that states the position of the item to be deleted, with 1 being the first position in the list.
- **from** – Keyword that must be included in the statement.
- **field** – Optional keyword identifying *window_field* as a field name.
- *window_field* – The name of the list field from which you'll delete the item specified in the *expression* parameter.

Comments

The field must be a window field, not a field in a table. If the position of the item to delete isn't known, the **finditem()** function can be used to find it.

If the currently-selected item is removed from the list field, the list field's value is set to zero. If one item is selected and another item is deleted, the currently-selected item will remain selected, regardless of whether its value has changed. For example, if the third item in a list is selected when the second item is deleted, the third item will remain selected, even though it's now in position 2 in the list.

Examples

The following example uses a number to identify the second item as the one to be deleted from the Payment Type field.

```
delete item 2 from 'Payment Type';
```

The following example uses the **finditem()** function to identify the item.

```
delete item finditem('Payment Type', "Open Item") from 'Payment Type';
```

Related items

Commands

add item, **countitems()**, **finditem()**, **insert item**, **itemname()**

delete line

Description

The **delete line** statement runs the delete line script associated with a scrolling window.

Syntax

delete line *scrolling_window_name*

Parameters

- *scrolling_window_name* – The name of the scrolling window from which the current line (based upon the contents of the table buffer) is to be deleted. The scrolling window must have a delete line script.

Comments

The delete line script for the scrolling window must contain the instruction to delete the item, typically a **remove** statement.

Examples

The following example runs the delete line script for the scrolling window Customer_Select_List.

```
delete line Customer_Select_List;
```

Related items

Commands

insert line

Additional information

Chapter 10, "Scrolling Windows," in Volume 2 of the Dexterity Programmer's Guide

delete table

Description

The **delete table** statement removes the named table from the operating system when the Pervasive.SQL or c-tree Plus database types are used. When a SQL database type is used, this statement either clears the data in a table and leaves the table structure intact, or removes the named table from the database entirely.

Syntax

delete table *table_name*

Parameters

- *table_name* – The name of the table to be deleted.

Comments

To use this statement, the table to be deleted must be opened by an **open table** command that includes the **exclusive** keyword.

This statement closes the named table before deleting it. The table is closed even if an error causes the table not to be deleted. Since you won't be notified of an error automatically, we recommend you use the **err()** function to find out whether an error occurred.

When the SQL database type is used, the entire table is removed from the database, unless the **Table_SetDeleteOptions()** function or the **SQLDropTableOnDelete** defaults file setting is used to change this functionality. If either the function or the defaults file setting is set to false, the **delete table** statement will remove the data from the named table, but leave the table structure intact in the database.

For applications using the SQL database type, Dexterity respects the access rights assigned to a table by the database administrator. If a user is granted read/write access only, Dexterity will return an error code if he or she attempts to use the **delete table** statement, since the user doesn't have delete authority.

Examples

The following example deletes the Customer_Master table.

```
open table Customer_Master, exclusive;
delete table Customer_Master;
if err() <> OKAY then
    warning "Unable to delete the table.";
end if;
```

Related items

Commands

open table, **Table_SetDeleteOptions()**

diff()

Description

The **diff()** function returns the difference between a field's new value and its old value.

Syntax

diff()

Parameters

- None

Return value

Numeric with the same type as the current field.

Comments

This function should be used only in field change scripts for numeric fields.

*This function should not be used in a change script run using the **run script** or **run script delayed** statements. Focus must be in the field for which this function is run for it function properly.*

Examples

The following example is a change script for the Total field that sets a local variable named net_change to the difference between the current and previous values of the Total field.

```
local integer net_change;

net_change = diff();
```

disable command

Description

The **disable command** statement disables a command. The command will appear disabled any place it appears, such as a menu or toolbar.

Syntax

disable command *command_name* {, *command_name*}

Parameters

- *command_name* – The name of the command to be disabled.

Comments

When a command is disabled, it cannot be executed by the user through the menus, toolbars or any shortcut key defined for the command. It also cannot be run through scripts with the **run command** statement.

Examples

The following example disables the CMD_HousesReport command. The command's appearance will be changed any place the command is displayed, such as a menu or toolbar.

```
disable command CMD_HousesReport;
```

The following example disables the CMD_Buyers and CMD_Sellers commands.

```
disable command CMD_Buyers, command CMD_Sellers;
```

Related items

Commands

enable command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

disable field

Description

The **disable field** statement disables one or more fields. When a field is disabled, the field is displayed in gray and can't be accessed by the user.

Syntax

disable field *field_name*{, *field_name*, *field_name*, ...}

Parameters

- *field_name* – The name of the field to be disabled. Multiple fields can be disabled by listing each field's name, separated by a comma. Because the fields can be in different windows, be sure to fully qualify the field names of fields not in the current window.

Comments

When the form or window is restarted, all disabled fields will revert to their original state as defined in the Properties window. However, if the field is disabled by the window pre script, it will be disabled again, as the window pre script runs each time the form or window are restarted.

The contents of a disabled field can still be manipulated by scripts.

*The **disable field** statement can't be used in scrolling windows to selectively disable one or more fields on certain lines. However, it can be used before the **fill window** statement to disable one or more fields on all lines of the scrolling window.*

Examples

The following example disables two fields using one statement. Since the Contact Name field is not located in the current window, its name is fully qualified.

```
disable field 'Customer ID', 'Contact Name' of window Contacts;
```

Related items

Commands

enable field

disable item

Description

The **disable item** statement disables a specific item in a button drop list.

Syntax

disable item *number* of **field** *name*

Parameters

- *number* – An integer indicating the position of the item that will be disabled in the button drop list.
- *name* – The name of the field for which an item is being disabled.

Examples

The following example disables the second item of the Reports button drop list field.

```
disable item 2 of field 'Reports';
```

Related items

Commands

enable item

Additional information

Button drop lists on page 65 of Volume 2 of the Dexterity Programmer's Guide

disable menu

Description

The **disable menu** statement disables a menu item. When a menu item is disabled, it is displayed in gray and can't be selected by the user.

Syntax

disable menu *menu_name number*

Parameters

- *menu_name* – The name of the menu that will have a command disabled.
- *number* – An integer representing the position of the item to be disabled within the menu.

Comments

Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, cannot be disabled. In addition, these inherited items should not be counted when determining the position of a menu item for use in the *number* parameter. For more information about menu items that can be disabled and enabled, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

When the form or window is restarted, all disabled menu items are reenabled, unless they are again disabled by the window pre script, which runs each time the form or window is restarted.

Examples

This example disables the third item in the Forms menu.

```
disable menu Forms 3;
```

Related items

Commands

enable menu

dow()

Description

The **dow()** function returns a value corresponding to the day of the week a date falls on.

Syntax

dow(*date***)**

Parameters

- *date* – A date or datetime value.

Return value

An integer between 0 and 6. The table lists the return values and the corresponding day of the week.

Value	Day
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Examples

The following example sets a local variable named this_date to the system date, then sets the value of a local variable named day_of_week to an integer corresponding to the day of the week.

```
local date this_date;
local integer day_of_week;

this_date = sysdate();
day_of_week = dow(this_date);
```

Related items

Commands

addmonth(), **day()**, **eom()**, **mkdate()**, **month()**, **setdate()**, **sysdate()**, **year()**

edit table

Description

The **edit table** statement reads or reserves a record and passively or actively locks it.

*If possible, use the **get** or **change** statements to retrieve information from a table. They're much easier to use.*

Syntax

edit table *table_name* { **by** *key_name* | **by number** *expr* } {, **lock**}{, **refresh**}

Parameters

- *table_name* – The name of the table containing the record you wish to edit.
- **by** *key_name* | **by number** *expr* – Identifies the key to use to search the table. 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.

*The key used by **edit table** must have a unique index (if a SQL database type is used) or not allow duplicates (if a Pervasive.SQL or c-tree Plus database type is used). Otherwise, you won't be able to retrieve records from the table using the **edit table** statement.*

- **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.

*The **lock** option shouldn't be used unless the Allow Active Locking option has been marked for the specified table in the Table Definition window. If the Allow Active Locking option hasn't been marked for a given table, any **edit table** statements specifying that table and including the **lock** option will not compile.*

  If your application will use the SQL database type, we recommend that you rely on passive locking instead of active locking. Using active locking with the SQL database type can cause performance degradation.

- **refresh** – This keyword only applies when a SQL database type is used. It indicates that the cursor and client-based buffer will be refreshed, so that they'll contain the most current data from the table before the edit takes place. Even if the edit attempt fails, the refresh will occur.

*If the Pervasive.SQL or c-tree Plus database types are used, the **refresh** keyword will be ignored. It will not affect your application in any way.*

Comments

The following sections describe how to use the **edit table** statement.

Preparation for edit table

The values of the key fields must be in the table buffer before the **edit table** statement is issued. Use the **clear table** statement to clear the other fields in the table buffer if they contain information from a previous transaction.

When the Pervasive.SQL and c-tree Plus database types are used, you must set up all non-modifiable key segments and set values for them before you issue an **edit table** statement. This must be done because the **edit table** statement saves all key values to the table; once these key values are saved, non-modifiable key segments cannot be changed. You can avoid this requirement by making only the primary key non-modifiable and making the rest of the keys modifiable when you define the table.

When a SQL database type is used, the key used by **edit table** statement must have a unique index created for it. Otherwise, you won't be able to retrieve records from the table using the **edit table** statement. Unique indexes are defined using the Keys window.

Execution of edit table

The **edit table** statement will attempt to write the record into the table using the key values in the table buffer.

If a record with those key values doesn't exist in the table, a new record will be added to the table, storing the key fields and any other fields that are set in the buffer. This is called *reserving a record*. This process creates the record so a record with the same key values can't be added or reserved by another user.

Once data has been placed in the other fields in the table buffer, use the **save table** statement to save the record. If a field has a default value established for it, and that field isn't set in the table buffer, the default value will be saved in the table. If the edit process is stopped – for instance, if the user decided to cancel the current entry – use the **release table** statement to remove the record from the table and release the lock.

If a record exists with those key values, the **edit table** statement will read the record and passively lock it. The **editexisting()** function will return true, indicating the record existed before the **edit table** statement was run. To test for an error during the retrieval, use the **err()** function.

You can use the **copy from table** statement to automatically transfer the information from the table buffer to windows in the form. This can be done only for the auto-copy fields. Use the **set** statement to transfer a single field of information from the table buffer to the window, or to transfer data from fields that aren't designated as auto-copy.

Examples

The following script is a change script for the Customer Number field in the Customer_Info window. It uses the **edit table** statement to either reserve a record in the table, or retrieve a record that already exists in the table. The **refresh** option ensures that, if the SQL database type is used, the most current data is accessed before the **edit table** statement is run.

```
{Release any lock from the previous operation.}
release table Customer_Master;
{Remove information from any previous operation.}
clear table Customer_Master;
{Set the key values before issuing the edit table statement.}
'Customer Number' of table Customer_Master = 'Customer Number' of
➥ window Customer_Info;
edit table Customer_Master by Cust_Number_key, refresh.
if editexisting() then
    {Customer exists, so retrieve the customer's information.}
    copy from table 'Customer_Master';
    {Otherwise, the new customer record has been reserved.}
end if;
```

The following is the change script for the Clear Button field in the Customer_Info window. If the current record has been reserved, but hasn't been saved, the script will remove the reserved record and restart the form.

```
release table Customer_Master;
restart form;
```

Related items

Commands

change, **editexisting()**, **release table**

Additional information

Chapter 15, "Working with Records," and *Chapter 18, "Multiuser processing," in Volume 2* of the Dexterity Programmer's Guide

editexisting()

Description

The **editexisting()** function returns a value indicating whether the last **edit table** statement retrieved an existing record or created and reserved space for a new record.

Syntax

editexisting()

Parameters

- None

Return value

A boolean. True indicates the record already existed and was retrieved. False indicates the record did not exist and space was reserved for it.

Examples

The following example ascertains whether an **edit table** statement was able to retrieve an existing record from a table named Customer_Master. If an existing record was retrieved, then the contents of the table buffer will be copied to the window buffer.

```
edit table Customer_Master;
if editexisting() then
    copy from table Customer_Master;
end if;
```

Related items

Commands

edit table

empty()

Description The **empty()** function returns a value indicating whether a field is empty.

Syntax **empty(***field_name***)**

Parameters

- *field_name* – The name of the field being checked.

Return value Boolean

Comments Numeric fields are considered empty if they contain a value of 0. String fields are considered to be empty if they contain nothing or only spaces. Date fields are considered empty when the date value is 0/0/0. Time fields are considered empty when the time value is 12:00:00 AM, which corresponds to 000000.

Examples The following example ascertains whether the Salesperson field contains a value. If it doesn't, a message will be displayed to the user.

```
if empty(Salesperson) then
    warning "No salesperson ID has been entered.";
end if;
```

enable command

Description

The **enable command** statement enables a command. The command will appear enabled any place it appears, such as a menu or toolbar.

Syntax

enable command *command_name* {, *command_name*}

Parameters

- *command_name* – The name of the command to be enabled.

Comments

When a command is enabled, it can be executed by the user through the menus, toolbars or the shortcut key defined for the command. It can also be run through scripts with the **run command** statement.

Examples

The following example enables the CMD_HousesReport command. The command's appearance will be changed any place the command is displayed, such as a menu or toolbar.

```
enable command CMD_HousesReport;
```

The following example enables the CMD_Buyers and CMD_Sellers commands.

```
enable command CMD_Buyers, command CMD_Sellers;
```

Related items

Commands

disable command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

enable field

Description

The **enable field** statement allows a previously disabled field to receive user input. The enabled field is redrawn using the settings specified in the Properties window for the field.

Syntax

enable field *field_name*{, *field_name*, *field_name*, ...}

Parameters

- *field_name* – The name of the field to be enabled. Multiple fields can be enabled by listing each field's name, separated by a comma. Because the fields can be in different windows, be sure to fully qualify the field names not in the current window.

Comments

Fields with the Editable property set to false in the Properties window *can't* be enabled using the **enable field** statement.

Examples

The following example enables fields in different windows. Since the Contact Name field isn't located in the current window, its name is fully qualified.

```
enable field 'Customer ID', 'Contact Name' of window Contacts;
```

Related items

Commands

disable field

enable item

Description

The **enable item** statement enables a specific item in a button drop list.

Syntax

enable item *number* of **field** *name*

Parameters

- *number* – An integer indicating the position of the item that will be enabled in the button drop list.
- *name* – The name of the field for which an item is being enabled.

Examples

The following example enables the second item of the Reports button drop list field.

```
enable item 2 of field 'Reports';
```

Related items

Commands

disable item

Additional information

Button drop lists on page 65 of Volume 2 of the Dexterity Programmer's Guide

enable menu

Description

The **enable menu** statement allows a previously disabled menu item to be selected by a user.

Syntax

enable menu *menu_name number*

Parameters

- *menu_name* – The name of the menu that will have an item enabled.
- *number* – An integer representing the position of the item to be enabled within the menu.

Comments

Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, cannot be enabled or disabled. In addition, these inherited items should not be counted when determining the position of a menu item for use in the *number* parameter. For more information about menu items that can be enabled and disabled, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

When the form or window is restarted, all disabled menu items are reenabled, unless they are again disabled by the window pre script, which runs each time the form or window is restarted.

Examples

This example enables the third item in the Forms menu.

```
enable menu Forms 3;
```

Related items

Commands

disable menu

endgroup

Description

The **endgroup** statement is used to indicate the end of a process group.

Syntax

endgroup *group_ID*

Parameters

- *group_ID* – A long integer variable representing a unique ID for this process group. The **begingroup** statement will generate this number and assign it to the variable named.

Comments

The **endgroup** statement must always follow a group of procedures set as a process group using the **begingroup** statement. Dexterity will not process a group until the **endgroup** statement is encountered.

Examples

See the example for **begingroup**.

Related items

Commands
begingroup, **call**, **call with name**

eom()

Description

The **eom()** function returns the date of the last day of the month for a specified date. For instance, if the date 07/21/94 is passed to the **eom()** function, it will return the date 07/31/94, the last day of the month for July. If the date is in a leap year, the last date in February will be adjusted accordingly.

Syntax

eom(*date***)**

Parameters

- *date* – A date or datetime value.

Return value

Date

Examples

The following example sets a local variable named last_day to the last day of the month for the month in the current date. The **sysdate()** function is used as the *date* parameter, so the operating system's current date is automatically passed to the **eom()** function.

```
local date last_day;

last_day = eom(sysdate());
```

Related items

Commands

addmonth(), **day()**, **dow()**, **mkdate()**, **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 wish to check the last error for.

Return value

Integer

Comments

The **err()** function is often used in an **if then...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 dialog box 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

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 P.SQL 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 for a save button attempts to save the current record to the Customer_Master table. The **err()** function is used to ascertain whether another user has changed the record. If the record has been changed, an error message is displayed and the record is read from the table again.

```
copy to table Customer_Master;
save table Customer_Master;
if err() = RECORDCHANGED then
    error "This record was changed by another user.";
    release table Customer_Master;
    change table Customer_Master;
    copy from table Customer_Master;
end if;
```

The following example is the change script for the Customer_ID field. It retrieves information related to the specific ID from two separate tables, Customer_Master and Customer_Data. It then checks whether any errors occurred during the retrieval process, displaying any successfully retrieved data in the Customer_Info window.

```
{Set the Cust_ID key of each table to the Customer_ID entered by the
user. Attempt to retrieve a record related to this ID from each
table.}
Customer_ID of table Customer_Master = Customer_ID of window
➥ Customer_Info;
Customer_ID of table Customer_Data = Customer_ID of  window
➥ Customer_Info;
change table Customer_Master by Cust_Master_CustID_key;
change table Customer_Data by Cust_Data_CustID_key;

{If either of the change table statements returned missing, check to
see if either of the statements did retrieve data. If so, copy that
information to the Customer_Info window, and prompt the user to add
the additional information.}
if err(table Customer_Master) = MISSING or err(table Customer_Data) =
➥ MISSING then
    if err(table Customer_Master) = OKAY then
        copy from table Customer_Master;
    elseif err(table Customer_Data) = OKAY then
        copy from table Customer_Data;
    end if;
    warning "Please enter the missing information for this
    ➥ customer.";
{If both successfully retrieved data, copy the data to the window.}
elseif err(table Customer_Master) = OKAY and err(table Customer_Data)
➥ = OKAY then
    copy from table Customer_Master;
    copy from table Customer_Data;
else
{Some problem occurred. Warn the user and clear the field and table
buffers.}
    warning "An error occurred while trying to retrieve information
    ➥ about this customer.";
    clear table Customer_Master;
    clear table Customer_Data;
    clear field Customer_ID;
end if;
```

Related items

Commands

check error, **naterr()**

Additional information

Chapter 18, "Multiuser processing," in Volume 2 of the Dexterity Programmer's Guide

error

Description

The **error** statement creates an error dialog box displaying the specified string.

Syntax

error *expression*{**with help number** *context_number*}

Parameters

- *expression* – A string field, text field, or string or text value with the message to be displayed in the dialog box.
- **with help number** *context_number* – A long integer specifying a help context number associated with a specific topic in the online help file for the current dictionary. If this parameter is used, a button labeled Help will appear in the dialog box. If a user presses the Help button, the specified help file topic will be displayed. Refer to Chapter 7, "Windows Help," in the Dexterity Stand-alone Application Guide for more information.

Comments

The dialog box will have one button labeled OK. The window closes automatically after the user clicks OK. The icon displayed is the standard error icon for the given operating system. The error dialog box is shown in the following illustration:

Examples

The following example displays an error message indicating a customer record couldn't be deleted. Note that a help context number (represented by the constant billion + 142) has been added. If the user clicks Help, the topic associated with that number will be displayed.

```
error "The customer " + 'Customer Name' + " has a balance and can't
➥ be deleted." with help number billion + 142;
```

Related items

Commands

ask(), **getmsg()**, **getstring()**, **substitute**, **warning**

execute()

Description

The **execute()** function allows you to compile and execute sanScript code at runtime. The code passed to the **execute()** function is called *pass-through sanScript*.

Syntax

execute ({*product_ID,*} *source*, *compile_error_message* {, *parameters*}**)**

Parameters

- *product_ID* – An optional integer that specifies the product ID of the dictionary in which you want to execute pass-through sanScript. This is useful if you want to access functionality in another application dictionary within the multidictionary environment.
- *source* – A string or text expression containing the sanScript code to compile and execute.
- *compile_error_message* – A returned string containing any compiler error that occurred when the sanScript code was compiled.
- *parameters* – Optional parameters to be passed to the sanScript code.

Return value

An integer indicating the number of compiling errors that occurred.

Comments

If you need to include a literal string in the pass-through sanScript code, you must enclose the string in two sets of quotation marks so the compiler will interpret the string properly.

When referencing resources in pass-through sanScript code, you must qualify the names based on which script will run the **execute()** function. For instance, if the **execute()** function will be run from a field change script, you must qualify all resource names in the pass-through sanScript code the same way you would qualify them in the field change script.

You can pass parameters into and out of pass-through sanScript code similar to the way you pass parameters into and out of procedures and functions. Simply declare the in, out or inout parameters at the beginning of the pass-through sanScript. Then pass the parameters to the script using the optional *parameters* for the **execute()** function.

By default, pass-through sanScript has access to the same table buffers as the script that runs the **execute()** function. For instance, if the **execute()** function is run from a field change script, the pass-through sanScript code will have the access to the same table buffers as that field change script. If the **execute()** function is run in a procedure or function script, the pass-through sanScript code will have access to its own private table buffers. If you want pass-through sanScript to access other table buffers, you must pass them in as parameters.

Examples

The following example is a field change script that executes pass-through sanScript to disable the Asking Price field.

```
local string compiler_error;

if execute("disable field 'Asking Price'.", compiler_error) <> 0 then
    {A compiler error occurred. Display the error.}
    error compiler_error;
end if;
```

The following example executes pass-through sanScript that displays a warning message. Notice that the literal string is enclosed in two sets of quotation marks.

```
local string compiler_error;

if execute("warning""This is a warning"".", compiler_error) <> 0 then
    {A compiler error occurred. Display the error.}
    error compiler_error;
end if;
```

The following example executes pass-through sanScript from a form-level procedure. Notice that the table buffer for the House_Data table is passed to the pass-through sanScript as a parameter. This means the pass-through sanScript will use the same table buffer as the form-level procedure. Otherwise, the pass-through sanScript would have its own private table buffer for the House_Data table.

```
local text code;
local string compiler_error;

{Build the pass-through sanScript code.}
code = code + "inout table House_Data.";
code = code + "warning 'House ID' of table House_Data.";

{Execute the code. Pass the House_Data table as a parameter.}
if execute(code, compiler_error, table House_Data) <> 0 then
    {A compiler error occurred. Display the error.}
    error compiler_error;
end if;
```

Related items

Commands
physicalname()

exit

Description

The **exit** statement allows you to exit the current instance of the specified program structure.

Syntax

exit [**for** | **repeat** | **while** | **try**]

Parameters

- **for** | **repeat** | **while** | **try** – A keyword indicating the type of program structure to exit from.

Comments

This statement is useful for navigation within nested loops. It is also useful for structured exception handling.

Examples

The following example uses the **exit** statement to exit a **while** loop.

```
local currency subtotal;
local integer inv_number;

subtotal = 0;
Customer_ID of table Customer_Orders = Customer_ID of table
➥ Customer_Info;
get table Customer_Orders by Cust_ID_Key;
while subtotal <= Credit_Limit of table Customer_Info do
    subtotal = subtotal + Order_Amount of table Customer_Orders;
    inv_number = invoice_number of table Customer_Orders;
    get next table Customer_Orders by Cust_ID_Key;
    if err() <> OKAY then
        warning "An error occurred after processing invoice " +
        ➥ inv_number;
        exit while;
    end if;
end while;
```

Related items

Commands

continue, **for do...end for**, **repeat...until**, **try...end try**, **while do...end while**

expand window

Description

The **expand window** statement switches the size of a scrolling window line between its normal line size and its expanded line size.

Syntax

expand window *scrolling_window* {**of form** *form_name*}, *boolean*

Parameters

- *scrolling_window* – The name of the scrolling window to be affected.
- **of form** *form_name* – An optional parameter specifying the form the scrolling window is part of.
- *boolean* – The value true sets the scrolling window to its expanded line size. The value false sets it to the normal line size.

Examples

The following example displays the scrolling window Customer List in its expanded line size.

```
expand window 'Customer List', true;
```

The following example returns the scrolling window Customer List to its normal line size.

```
expand window 'Customer List', false;
```

Related items

Additional information

Chapter 10, "Scrolling Windows," in Volume 2 of the Dexterity Programmer's Guide

extern

Description

The **extern** statement calls a procedure containing the name and parameter set for a DLL (Dynamic Link Library).

Syntax

extern *procedure*, *return_value*{, *parameter_list*}

Parameters

- *procedure* – The name of the prototype procedure containing the name and parameter set for the DLL. The prototype procedure name must include the name of the function within the DLL, the @ symbol, and be followed by the name of the DLL to be called. This parameter must be enclosed in single quotation marks; it will not compile properly without them.
- *return_value* – A field that will be set to the value returned by the specified DLL. The control type of this field must match that of the DLL's return value.
- *parameter_list* – The list of parameters to be sent to and received from the DLL.

Comments

You must create a prototype procedure before you can call a DLL. This is described in Chapter 22, "Dynamic Link Libraries," in Volume 2 of the Dexterity Programmer's Guide.

Examples

The following example calls the function named MessageBox in the DLL named USER32.DLL. The return value from the function will be stored in the local long integer variable named result.

```
local long result, window_handle;
local string text_string, title_string;
local integer style;

text_string = "This is a DLL test.";
title_string = "DLL Test";
style = 0. {OK style}
{Set window_handle to indicate that there is no parent window.}
window_handle = 0;
extern 'MessageBox@USER32.DLL', result, window_handle, text_string,
➥ title_string, style;
```

Related items

Additional information

Chapter 22, "Dynamic Link Libraries," in Volume 2 of the Dexterity Programmer's Guide

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:

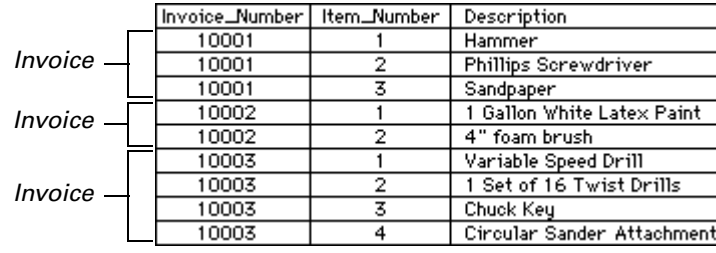
Storage type	Value
Date	12/31/9999
Currency	99999999999999.99999
Integer	32,767
Long	2,147,483,647
String	The length byte (first byte) of the string is set to the storage size of the string minus 1. Each of the remaining bytes is set to string equivalent of ASCII 255.
Time	23:59:59

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.

Examples

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;
Invoice_Number = 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, **fill table**, **filled()**, **range**

Additional information

Chapter 16, "Ranges," in Volume 2 of the Dexterity Programmer's Guide

fill table

Description

The fill table statement sets every field in the table buffer for the specified table to the largest value represented by the field's data type. For example, an integer field would be filled with the number 32,767.

Syntax

fill table *table_name*

Parameters

- *table_name* – The name of the table whose table buffer fields will be filled.

Comments

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

The following table lists the storage types for which the fill operation applies, and the value to which the field will be set.

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
DateTime	12/31/9999 23:59:59

If the table specified is not open, the **fill table** statement will open it.

Examples

The following example sets a range for the House_Data table. It uses the **clear table** and **fill table** statements to clear and fill all of the fields in the table buffer when setting the range.

```
range clear table House_Data;
clear table House_Data;
'Seller ID' of table House_Data = "1000";
range start table House_Data by number 2;
fill table House_Data;
'Seller ID' of table House_data = "1000";
range end table House_Data by number 2;
```

Related items

Commands

clear field, **clear table**, **fill**, **filled()**, **range**

fill window

Description

The **fill window** statement fills a scrolling window from the linked table or another specified table. You can specify a starting place for the fill, and the key by which records will be sorted in the scrolling window.

Syntax

fill window *name* { **from top** | **from current** | **from bottom** | **redraw {up | down}** } { **table** *table_name* | **using current table** } { **by** *key_name* | **by number** *expr* }

Parameters

- *name* – The name of the scrolling window to be filled.
- **from top** | **from current** | **from bottom** | **redraw {up | down}** – The starting position for the window fill. Selecting **from top** fills the scrolling window with records from the top of the selected range. Selecting **from current** fills the scrolling window with records starting from the record currently in the table buffer. This record will appear at the top of the scrolling window. Selecting **from bottom** fills the scrolling window with records from the bottom of the selected range. Selecting **redraw** redraws the scrolling window without causing any of the displayed items to scroll. If no starting point is specified, the window will be filled from the top or bottom of the range, depending upon how the scrolling window's ScrollToBottom property is set.

  The optional **up** or **down** keywords for the **redraw** clause are used when a record currently visible in the scrolling window is moved to a different position in table used for the scrolling window. If you want to redraw the scrolling window to show the moved record's new location, you need to indicate whether the record moved toward the beginning of the table (up) or toward the end of the table (down). This allows Dexterity to redraw the scrolling window and keep the moved record visible.

- **table** *table_name* | **using current table** – Allows you to specify another table to fill the scrolling window from, instead of the linked table. If this parameter isn't included, the linked table will be used to fill the window. You can explicitly specify the table to use with the **table** *table_name* clause. If you use the **using current table** clause, the table currently being used to fill the scrolling window will continue to be used.

- **by** *key_name* | **by number** *expr* – Identifies the key to use when adding records from the table to the scrolling window. If one of these parameters isn't included, the key specified by the scrolling window's LinkTableKey property 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.

Examples

The following example fills the Customer List scrolling window using the key Name_Key, starting at the current record.

```
fill window 'Customer List' from current by Name_Key;
```

The following example fills the scrolling window Customer List from the table New_Customers, using the first key. When filled, the window will display the top or bottom of the range, depending upon how the scrolling window's ScrollToBottom property is set.

```
fill window 'Customer List' table New_Customers by number 1;
```

Related items

Additional information

Chapter 10, "Scrolling Windows," in Volume 2 of the Dexterity Programmer's Guide

filled()

Description

The **filled()** function returns true if the specified field is set to its maximum value; otherwise, it returns false.

Syntax

filled(*field_name***)**

Parameters

- *field_name* – The name of the field.

Return value

Boolean

Comments

The following table lists the storage types for which the **filled()** function can be used, and the value that each type will contain when it is considered 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 will be the storage size of the string minus 1. Each of the remaining bytes will be set to string equivalent of ASCII 255.
Time	23:59:59
DateTime	12/31/9999 23:59:59

If the **filled()** function is used to check a field in a table that isn't open, the table will be opened.

Examples

The following example uses the **filled()** function to verify whether a table field was set to its maximum value using the fill statement.

```
if filled(Item_Number of table Invoice_Data) then
    Item_Display = "All Line Items";
else
    Item Display = "Single Line Item";
end if;
```

Related items

Commands

clear field, **fill**, **range**

finddata()

Description

The **finddata()** function returns the numeric position of a data item in a list field that is associated with a specified numeric value. If no data item is associated with the specified value, 0 is returned.

Syntax

finddata(*window_field*, *value***)**

Parameters

- *window_field* – A list field displayed in the window.
- *value* – A long integer previously associated with an item in the list field using the **add item** statement.

Return value

Integer

Comments

The **add item** statement allows you to add items to a list field and associate numeric values with those items. While these values aren't displayed in the list field, the **finddata()** function will return the position in the list of a specified value and its related item. For example, if an item added was the fifth item in a given list, and its associated value was 1234, specifying that list field name and 1234 as the parameters in a **finddata()** function will return the number 5.

Values can be associated only with items that are added to a list field using the **add item** statement. You can't associate numeric values with the static values assigned to a list field's data type. If the same numeric value is associated with two or more static values, the position of the first matching value will be returned.

Examples

The following example sets the value of the customer_number local variable to the position of the Customer ID value 1047, associated with a customer name which was added to the Customers list field.

```
local integer customer_number;

customer_number = finddata(Customers, 1047);
if customer_number = 0 then
    warning "The item isn't in the list.";
end if;
```

Related items

Commands

add item, **countitems()**, **delete item**, **finditem()**, **insert item**, **itemdata()**, **itemname()**

finditem()

Description

The **finditem()** function returns the numeric position of a specific item in a menu or list field. List fields include list boxes, multi-select list boxes, combo boxes, drop-down lists, button drop lists and visual switches. If the specified item doesn't appear in the menu or field, a 0 is returned.

Syntax

finditem([**field** / **menu**] *name*, *expression***)**

Parameters

- *name* – The name of the menu or list field for which you want to return the position of an item. The field must be a window field.
- *expression* – A string specifying the item you want to find.

Return value

Integer

Comments

Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, are not counted when the location of an item in a menu is returned.

Examples

The following example sets the value of a local variable named item_number to the position of the UPS Red text value in the Shipping Methods field. If the entry isn't found, a warning message will be displayed indicating the item wasn't found in the list.

```
local integer item_number;

item_number = finditem('Shipping Methods', "UPS Red");
if item_number = 0 then
    warning "The item isn't in the list.";
end if;
```

The following example retrieves the location of the 20% item in the Markup menu and disables it.

```
disable menu Markup, finditem(menu Markup, "20%");
```

Related items

Commands

add item, **countitems()**, **delete item**, **finditem()**, **itemname()**

Additional information

Chapter 6, "Data Types," in Volume 1 of the Dexterity Programmer's Guide

focus

Description

The **focus** statement moves the focus to the specified field. This statement can be used to skip unnecessary fields in a window, or to place the focus in a specific field after an event.

Syntax

focus {**field**} *field_name*

Parameters

- **field** – An optional keyword identifying the following name as a field.
- *field_name* – The name of the field to which the focus is to be moved.

Examples

The following example places the focus on the Discount Percentage field.

```
focus field 'Discount Percentage';
```

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 or long 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 specifying 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.

Examples

In the following example, the monthly sales amounts for a year are stored in the Yearly Sales array field. The for loop sets these amounts to zero. Note that the loop index variable "i" is used as the array index.

```
{Set up the loop index.}
local integer i;

for i = 1 to 12 do
    'Yearly Sales'[i] = 0;
end for;
```

Related items

Commands

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

force change

Description

The **force change** statement causes the field change script to be run when the user moves the focus out of the field, whether or not the field has changed. This statement should be used only for the field where the focus is currently positioned.

Syntax

force change *field_name*

Parameters

- *field_name* – The name of the field for which you wish to run the change script, and where the focus is currently positioned.

Comments

The **force change** statement is typically placed in the pre script for a field.

Examples

The following example forces the change script for the Beginning Balance field to run.

```
force change 'Beginning Balance';
```

Related items

Commands

clear force change

force changes

Description

The **force changes** statement sets the change flag for the specified window or form. This flag is tested by the **changed()** function.

Syntax

force changes [**form** | **window**] *name* {**of form** *form_name*}

Parameters

- **form** | **window** – Indicates the type of the object for which you want to set the change flag.
- *name* – The name of the window or form for which the change flag will be set.
- **of form** *form_name* – If setting the change flag for a window, an optional parameter specifying the form the window is part of.

Comments

You can use the **force changes** statement with a scrolling window to force the line change script to run, regardless of whether fields in the scrolling window line have changed.

Examples

The following example sets the change flag for the Customer_Maintenance form.

```
force changes form Customer_Maintenance;
```

The following example sets the change flag for the Customer_Comments window of the Customer_Maintenance form.

```
force changes window Customer_Comments of form Customer_Maintenance;
```

Related items

Commands

clear changes, **changed()**

format()

Description

The **format()** function converts the contents of a currency or variable currency field to a string, applying the specified formatting.

Syntax

format(*currency, cur_symbol*, *thou_sep*, *dec_#*, *type***)**

Parameters

- *currency* – A currency or variable currency field or value you want to format and convert to a string.
- *cur_symbol* – A boolean value indicating whether the currency symbol will be included with the currency amount. True indicates that the symbol will be included; false indicates that no symbol will be used.
- *thou_sep* – A boolean value indicating whether the thousands separator will be included with the currency amount. True indicates that the separator will be used; false indicates that no separator will be used.
- *dec_#* – An integer value indicating the number of places that appear to the right of the decimal separator. For currency fields, this must be between 0 and 5. For variable currency fields, it must be between 0 and 15.

  If this value is greater than or equal to 1000, the multiformat value associated with the specified integer will be applied to the item indicated by the *currency* parameter. If the *cur_symbol* and *thou_sep* parameters are set to true, the currency symbol and thousands separator defined for the multiformat value will be used.
- *type* – One of the four constants shown below, indicating how a negative currency should be displayed. To display a percentage sign at the end of the number, add 100 to the constant (for instance, MINUSNEG+100).

Constant	Description
SYSTEMNEG	The international settings in the operating system.
MINUSNEG	Minus sign
PARENNEG	Parentheses
CRNEG	CR

Return value

String

Comments

The currency symbol and thousands separator used with this function are the operating system defaults, not any symbols specified for particular multiformat values using the functions in the Currency function library.

Examples

The following example uses the value in the Amount field and converts it to a string, changing the way the currency value is formatted by adding the currency symbol, a comma as the thousands separator, displaying only two decimal places, and using a minus sign for any negative amounts. The result is stored in the local variable l_result.

```
local string l_result;

{Value in Amount field is displayed as 12284.5345.}
l_result = format(Amount ,true, true, 2, MINUSNEG);
{Value is stored in l_result as the string value $12,284.53}
```

In the following example, the multiformat value 1000 was defined using functions from the Currency function library. The *dec_#* parameter indicates that this multiformat value will be used when formatting the value from the Currency Amount field. The format will include the currency symbol, thousands separator and negative symbol defined for the multiformat value.

```
local string currency_str;

currency_str = format('Currency Amount', true, true, 1000, MINUSNEG);
```

Related items

Additional information

The Currency function library in the Function Library Reference manual

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**}{**with**(*hint*)}

Parameters

- **next** | **prev** | **first** | **last** – Identifies which record you wish 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* – The name of the table containing the record to read.

- **by** *key_name* | **by number** *expr* – Identifies the key to use 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.

- **refresh** – This keyword only applies when a SQL database type is used. It indicates that the cursor and client-side buffer will be refreshed, so that they'll contain the most current data from the table before the **get** statement is used. Even if the **get** attempt fails, the refresh will occur.

*If the Pervasive.SQL or c-tree Plus database types are used, the **refresh** keyword will be ignored. It will not affect your application in any way.*

- **with**(*hint*) – This keyword applies only when a SQL database type is used. It allows you to supply an optimizer hint to control how the SQL Server query optimizer works. In this version of Dexterity, the following optimizer hints are supported:

Hint	Description
NOLOCK	Specifies that the SQL Server should not issue a shared lock and to disregard any existing exclusive locks on the row being read. This is referred to as a "dirty read".
FORCEINDEX	Causes the SQL Server to disregard the query plan it generated and use the index specified by the **get** statement. If no key has been specified for the **get** statement, the primary key will be used. If a clustered index exists for the table, specifying that the **get** statement use key 0 forces a clustered index scan. If no clustered index exists, using key 0 will force a table scan.

Comments

The values of the key fields should be set in the table buffer before the **get** 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.

The **get** statement fills only the table buffer. To transfer information to the window so the information can be displayed, use a **copy from table** statement. To transfer a single field of information from the table buffer to the window, use the **set** statement.

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 may have occurred.

Examples

The following example attempts to retrieve the account description after the table buffer has been set to an account number entered by the user. The **refresh** option is included to ensure that, if a SQL database type is used, the most current data will be retrieved.

```
'Account Number' of table GL_Accounts = 'Account Number' of window
➥ AP_Distributions;
get table GL_Accounts by Account_key, refresh;
if err() = OKAY then
    'Account Description' of window AP_Distributions =
    ➥ 'Account Description' of table GL_Accounts;
    'Account Type' of window AP_Distributions = 'Account Type' of
    ➥ table GL_Accounts;
else
    warning "The account could not be located.";
end if;
```

The following example reads a record from a SQL table. It uses an optimizer hint to prevent the SQL Server from locking the record before it can be read.

```
get last table Note_Master by key 1 with(NOLOCK);
```

The following example reads a record from a SQL table and uses an optimizer hint to force the SQL Server to use the second index for the table.

```
get table User_Master by key 2 with(FORCEINDEX);
```

Related items

Commands

check error, **err()**

Additional information

Chapter 15, "Working with Records," in Volume 2 of the Dexterity Programmer's Guide

getfile()

Description

The **getfile()** function creates a dialog box that allows the user to select a file. It returns a boolean value indicating whether the user clicked OK or Cancel in the dialog box. If the user clicks OK or Open, the file name and path will be returned to the variable named in the *path* parameter.

The files displayed in the dialog can be filtered so that only certain types of files are displayed. You can use a predefined filter, or create your own filter.

Syntax

getfile(*prompt*, *default_filter*, *path* {, *custom_filter*}**)**

Parameters

- *prompt* – A string that will be displayed in the title bar of the dialog box.
- *default_filter* – An integer indicating the type of file you're searching for. If you are using one of the predefined filters, this value corresponds to one of the following constants:

Constant	Description
APPFILE	Displays only application files
DICTIONARY	Displays only dictionary files (.DIC)
LAUNCH	Displays only launch files (.SET)
MACROFILE	Displays only Dexterity macro files (.MAC)
TEXTFILE	Displays all files

  If you are using a custom filter, the *default_filter* parameter indicates which filter in the custom filter set will be used by default. For instance, assume three filter types are defined in the custom filter. To specify that the second filter type be used as the default, use 2 for this parameter.

- *path* – A string variable to which the file path and name will be returned. The returned path will be in generic format. You can supply a default value to be displayed in the file dialog by setting the *path* variable before opening the dialog. If you supply only a filename, the current path will be used. If you supply a complete path (in native format) the default location will also be set in the dialog.
- *custom_filter* – An optional string that specifies a set of custom filters to use. The set of custom filters is defined using the following syntax:

  description |*filter*{;*filter*}|

The *description* parameter is a string that describes the filter, such as "Dictionaries (*.DIC)". It must be followed by a pipe (|) character.

The *filter* parameter specifies a file extension, such as "*.DIC". When this filter is used, files that match this extension will be displayed in the dialog box. The *filter* parameter should not contain spaces. To specify multiple filters, separate them with a semicolon (;). End the list of filters with a pipe (|) character.

Several custom filters can be concatenated within the *custom_filter* string.

Return value

A boolean value indicating which button was clicked. If Open is clicked, the value of true is returned, and the file name and path (a string value) are returned to the variable named in the *path* parameter. If Cancel is clicked, the value false is returned.

Comments

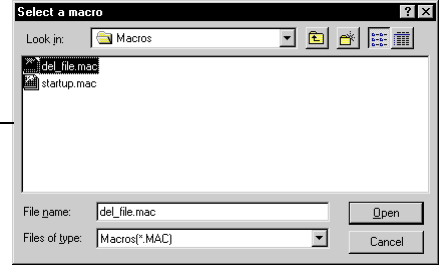
If you will be using custom filters, consider creating messages or constants for each type of filter. You can then easily concatenate the filter types together and build the *custom_filter* string. This is shown in the fourth example.

Examples

The following example retrieves a file path and name of a macro file to be run. The macro will be run only if the user clicks OK. The MACROFILE constant is used so that only macro files will be displayed to the user.

```
local string macro_name;

if getfile("Select a macro", MACROFILE, macro_name) then
    run macro macro_name;
end if;
```



*The **getfile()** function in the previous example would display this dialog box.*

The following example retrieves a file path and the name of a launch file from which information will be retrieved. The **Launch_CountProds()** function is then used to retrieve the number of products from the selected launch file:

```
local string l_launch_file;
local integer l_products;

if getfile("Select a launch file", LAUNCH, l_launch_file) then
     {Return the number of products in the launch file.}
     l_products = Launch_CountProds();
end if;
```

The following example sets the default location for the dialog to open the SampleData.mac macro file.

```
local string macro_name;

{Set the default path}
macro_name = "C:\Macros\SampleData.mac";

if getfile("Select the sample data macro", MACROFILE, macro_name) then
    run macro macro_name;
end if;
```

The following example uses a custom filter so that only Microsoft Word documents are displayed. The description for the custom filter is "Word Documents (*.doc;*.dot;*.rtf)". The custom filter will display three types of files: "*.doc", "*.dot", and "*.rtf". Notice that the filters are separated by semicolons, and that the custom filter ends with a pipe (|) character. By default, the first custom filter will be used.

```
local boolean result;
local string path;

result = getfile("Select a file", 1, path,
➥ "Word Document (*.doc;*.dot;*.rtf)|*.doc;*.dot;*.rtf|");
```

The following example shows how constants can be used to create custom filters. The following constants have been added to the current dictionary:

WORD_FILES	**Word (*.doc;*.dot;*.rtf)|*.doc;*.dot;*.rtf|**
EXCEL_FILES	**Excel (*.xls;*.xlw)|*.xls;*.xlw|**
PPT_FILES	**PowerPoint (*.ppt;*.pps)|*.ppt;*.pps|**

The following script uses these constants to create a custom filter for the **getfile()** function:

```
local boolean result;
local string path;

result = getfile("Select a document", 1, path, WORD_FILES +
➥ EXCEL_FILES + PPT_FILES);
```

Related items

Commands
Path_SelectPathname()

getmsg()

Description

The **getmsg()** function returns a message string created with the Messages resource.

Syntax

getmsg(*message_ID***)**

Parameters

- *message_ID* – An integer containing the message ID of the message you wish to retrieve.

Return value

String

Examples

The following example uses the **getmsg()** function and the **warning** statement to display a message with the ID 5.

```
warning getmsg(5);
```

Related items

Commands

error, **substitute**, **warning**

Additional information

Chapter 18, "Messages," in Volume 1 of the Dexterity Programmer's Guide

getstring()

Description

The **getstring()** function creates a dialog box that allows the user to enter a string, and returns a boolean value indicating whether the user clicked OK or Cancel in the dialog box. When the dialog is closed, the string will be returned in the variable named in the *string_variable* parameter.

Syntax

getstring(*prompt*, *password*, *string_variable* {, *context_number*}**)**

Parameters

- *prompt* – A message string that will be displayed in the dialog box.
- *password* – A boolean value indicating how the entry will be displayed. If set to true, the entry will appear as Xs, preventing others from seeing what is being entered. If set to false, the user's entry will appear normally.
- *string_variable* – If this variable contains a value, it will appear as a default when the dialog is displayed. When the user closes the dialog by clicking OK or Cancel, the value in the dialog will be returned to this variable.
- *context_number* – An optional long integer representing a help context number associated with a specific topic in the online help file for the current dictionary. If this parameter is used, a button labeled Help will appear in the dialog box. If a user presses the Help button, the specified help file topic will be displayed. Refer to Chapter 7, "Windows Help," in the Dexterity Stand-alone Application Guide for more information.

Return value

A boolean. If OK was clicked, the value of true is returned, and the string entered by the user is returned to the *string_variable* parameter. If Cancel was clicked, only the boolean value false is returned.

Examples

The following example prompts the user to enter a password. If the entry is valid, the Customers window will be opened.

```
local string user_entry;

if getstring ("Please enter password", true, user_entry)then
    if user_entry = 'Customer Password' of table Passwords then
        open window Customers;
    end if;
end if;
```

havetransactions()

Description

The **havetransactions()** function will return true if the current default database type has transaction processing capabilities; otherwise, it will return false.

Syntax

havetransactions()

Parameters

- None

Return value

Boolean

Examples

The following example ascertains whether the current database type has transaction processing capability. If it does, the procedure named Transaction_Check_Posting will be called to post checks. Otherwise, the Check_Posting procedure will be called to post checks.

```
if havetransactions() then
    {Use transaction processing.}
    call Transaction_Check_Posting;
else
    {Don't use transaction processing.}
    call Check_Posting;
end if;
```

Related items

Commands

transaction begin, **transaction commit**, **transaction rollback**

Additional information

Chapter 19, "Transactions," in Volume 2 of the Dexterity Programmer's Guide

hex()

Description

The **hex()** function returns a string containing the hexadecimal representation of an integral value.

Syntax

hex(*value***)**

Parameters

- *value* – The integral value which will be returned in hexadecimal form.

Return value

A string containing the hexadecimal equivalent of the integral value supplied.

Comments

The hexadecimal value will have the form &h*nnnnnnnn*. The hexadecimal digits A to F will be returned as lower case. You can convert them to upper case using the **upper()** function.

Examples

The following example sets the string hex_string to the hexadecimal equivalent of the integral value 200.

```
local string hex_string;

hex_string = hex(200);
{hex_string is "&h00c8"}
```

Related items

Commands

value()

Additional information

Hexadecimal values on page 55 of Volume 2 the Dexterity Programmer's Guide

hide command

Description

The **hide command** statement makes the named command invisible every place it is displayed, such as a menu or toolbar.

Syntax

hide command *command_name* {,*command_name*}

Parameters

- *command_name* – The name of the command to be hidden.

Comments

When a command is hidden, it can still be executed through scripts with the **run command** statement. It can also be accessed using the shortcut key defined for the command. To make a command inaccessible, use the **disable command** statement.

Examples

The following example hides the CMD_HousesReport command. Any place the command is displayed, such as a menu or toolbar will be hidden.

```
hide command CMD_HousesReport;
```

The following example hides the CMD_Buyers and CMD_Sellers commands.

```
hide command CMD_Buyers, command CMD_Sellers;
```

Related items

Commands

show command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

hide field

Description

The **hide field** statement makes the named field or fields invisible and inaccessible to the user.

Syntax

hide field *field_name* {, *field_name*, *field_name*...}

Parameters

- *field_name* – The name of the field to be hidden. Multiple fields can be hidden by listing the field names, separated by commas.

Comments

Hidden fields will appear to have been removed from the screen, but they can still be accessed from a script. To make the field usable again, use the **show field** statement.

*The **hide field** statement can't be used in scrolling windows to selectively hide a field on certain lines. It can be used before the **fill window** statement to hide a field on all lines of the scrolling window.*

Examples

The following example hides the Discount Percentage field.

```
hide field 'Discount Percentage';
```

The following example hides the Discount Percentage and Calculation Method fields.

```
hide field 'Discount Percentage', 'Calculation Method';
```

Related items

Commands
show field

hide menu

Description

The **hide menu** statement hides the specified menu item.

Syntax

hide menu *menu_name*, *number*

Parameters

- *menu_name* – The name of the menu containing the item you want hidden.
- *number* – An integer representing the placement of the item to be hidden in the menu.

Comments

The **hide menu** statement cannot be used to hide grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste. For more information about menu items that can be hidden and shown, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

Examples

The following example hides the third item in the Transactions menu.

```
hide menu Transactions, 3;
```

Related items

Commands

show menu

hide window

Description

The **hide window** statement makes the specified scrolling window invisible and inaccessible to the user.

Syntax

hide window *window_name*

Parameters

- *window_name* – The name of the scrolling window to hide.

Comments

When hiding a scrolling window, any fields or drawn objects that directly or indirectly touch the scrolling window will also be hidden.

Examples

The following example makes the Customer_Entry_Scroll scrolling window invisible.

```
hide window Customer_Entry_Scroll;
```

Related items

Commands

show window

hour()

Description

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

Syntax

hour(*time***)**

Parameters

- *time* – A time or datetime value

Return value

An integer between 0 and 23.

Examples

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;

hour_of_time = 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 of these clauses is 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** can be included in an **if then...end if** statement.
- *statements* – any valid sanScript statement or statements.

Examples

The following example places the focus on the Purchase Amount field if the Credit Line field is greater than 0.

```
if 'Credit Line' > 0 then
    focus 'Purchase Amount';
end if;
```

The following example uses an **elseif** clause to set the correct type of invoice, based upon the customer type. If the customer type isn't 1 or 2, a warning is displayed to alert the user.

```
if 'Customer Type' of table Customer_Master = 1 then
    'Invoice Type' = 1;
elseif 'Customer Type' of table 'Customer Master' = 2 then
    'Invoice Type' = 2;
else
    warning "Customer type must be 1 or 2.";
end if;
```

Related items

Commands
case...end case

import

Description

The **import** statement creates a reference to the specified type library file, allowing the current script to access the methods, properties, and other items described in the type library.

Syntax

import *library*

Parameters

- *library* – A string containing the complete path to the type library file, or the GUID that uniquely identifies the type library. If the file is located in the search path for the current workstation, you don't need to specify the complete path to the type library file. To use the GUID, the type library must be registered on the current workstation.

Comments

Typically, you would use the **import** statement only if you were using the COM functionality in a limited number of scripts in your application, or were using pass-through sanScript. In all other cases, we recommend creating a type library reference for the entire dictionary.

If you are specifying a type library based on the GUID, you can also indicate which version of the type library to use. After the GUID in the string for the *library* parameter, append a comma and the version number of the library. If the specified version of the library can't be found, a compiler error will occur.

Examples

The following example imports the type library for Adobe Acrobat. Notice that the complete path to the type library file is used.

```
import "C:\Program Files\Adobe\Acrobat 4.0\Acrobat\Acrobat.TLB";
```

The following example uses the GUID to import the Adobe Acrobat type library. The type library must be registered for the current machine, or the script will not compile.

```
import "{9B4CD3E5-4981-101B-9CA8-9240CE2738AE}";
```

The following example uses the GUID to import version 1.0 of the Adobe Acrobat type library.

```
import "{9B4CD3E5-4981-101B-9CA8-9240CE2738AE},1.0";
```

Related items

Additional information
Chapter 37, "Libraries," in Volume 2 of the Dexterity Programmer's Guide

increment

Description

The **increment** statement allows you to increment, or increase, a numeric value by a specified amount.

Syntax

increment *number* {**by** *numeric_expression*}

Parameters

- *number* – An integer, long, currency, variable currency, time or date field or variable that you want to increment.
- *numeric_expression* – A field or variable of the same control type as the *number* parameter, by which *number* is increased. If no *numeric_expression* is indicated, the default increment value is 1. The following default values are supported for each numeric control type:

Control type	Default increment value
integer	1
long	1
currency	1 currency unit
variable currency	1 currency unit
time	1 minute
date	1 day

Examples

The following example increases an existing sequence number by five.

```
increment 'Sequence Number' by 5;
```

In the following example, no numeric expression is indicated, so the resulting sequence number is increased by 1.

```
increment 'Sequence Number';
```

The following example increases a date value by a numeric value ('Number of Days to Pay'). If the 'Number of Days to Pay' is 16, the resulting date is 16 days in the future.

```
increment 'Invoice Date' by 'Number Of Days To Pay';
```

Related items

Commands

decrement

insert item

Description

The **insert item** statement inserts an item into the specified position in a list field. List fields include list boxes, multi-select list boxes, combo boxes, drop-down lists, button drop lists or visual switches. This statement also allows you to associate a numeric value with the item inserted into the list.

Syntax

insert item *string_expression* {, *value*} **in** {**field**} *window_field* **at** *position*

Parameters

- *string_expression* – The string you want to insert into the list field.
- *value* – An optional parameter containing the long integer to add to the list field in conjunction with the *string_expression*. If this parameter isn't used, 0 will be associated with the *string_expression*.
- **field** – An optional keyword identifying *window_field* as a field.
- *window_field* – A list field displayed in the window.
- **at** *position* – An integer indicating the visible position into which the item will be inserted.

Comments

The new item will be inserted at the position specified, and also have that same value. Any other items in the list that have values greater than the insert position for the inserted item will have their values increased by one. For example, assume the **insert item** statement is used to insert a new item at position 3. The new item will have the value 3. The previous item that had the value 3 will now have the value 4. The item that had the value 4 will now have the value 5, and so on.

If the value of the item selected in the list field is greater than the position into which the new item was inserted, the value of the list will increase by one. For example, assume the item with the value 4 is selected in a list field. If a new list item inserted into position 3, the value of the list field will increase to 5, because the value of the selected item was increased by one.

If items are inserted while the field is displayed, the field must be redrawn before the added string will appear.

The numeric values related to the added items won't appear in the list. However, they can be retrieved using the **itemdata()** function, and then used in scripts.

Examples

The following is part of a window pre script. It adds the item "Gift Certificate" as the second item in the Payment Method list box.

```
insert item "Gift Certificate" in field 'Payment Method' at 2;
```

Related items

Commands

add item, change item, delete item, finddata(), finditem(), itemname(), redraw, Field_GetInsertPosFromVisualPos(), Field_GetVisualPosFromInsertPos()

insert line

Description

The **insert line** statement runs the insert line script associated with a scrolling window.

Syntax

insert line *scrolling_window_name*

Parameters

- *scrolling_window_name* – The name of the scrolling window into which a line will be inserted. The scrolling window must have an insert line script.

Comments

The insert line script for the scrolling window must contain the instructions to add an item, typically a **save table** statement.

Examples

The following example runs the insert line script for the Customer_Select_List scrolling window.

```
insert line Customer_Select_List;
```

Related items

Commands

delete line

Additional information

Chapter 10, "Scrolling Windows," in Volume 2 of the Dexterity Programmer's Guide

isalpha()

Description

The **isalpha()** function indicates whether a string expression contains only alphabetic characters.

Syntax

isalpha(*string_expression* {, *allow_spaces*}**)**

Parameters

- *string_expression* – The string that will be examined to find out whether it contains only alphabetic characters.
- *allow_spaces* – An optional boolean parameter that specifices whether spaces are allowed in *string_expression*. The value true indicates spaces are considered alphabetic characters, while false indicates they are not. The default value is true.

Return value

A boolean. True indicates the entire string expression is composed of alphabetic characters, while false indicates it is not.

Comments

The **isalpha()** function returns true if the empty string ("") is passed in as the string expression.

Examples

The following examples show how string expressions are evaluated to find out whether they contain only alphabetic characters.

```
local boolean result;

result = isalpha("Dexterity");           {result = true}
result = isalpha("Dexterity6");          {result = false}
result = isalpha("Great Plains");        {result = true}
result = isalpha("Great Plains", false); {result = false}
result = isalpha("");                    {result = true}
```

isopen()

Description

The **isopen()** function returns a value indicating whether a specified form or window is open.

Syntax

isopen([**form** | **window**] *name***)**

Parameters

- **form | window** – Indicates whether you're checking the status of a form or window.
- *name* – The name of the form or window you're checking.

Return value

A boolean value that is set to true if the specified form or window is open. False is returned if the specified form or window isn't open.

Examples

The following example checks whether the Customer_Maintenance window is open. If the window is open, a message to that effect will be displayed to the user. If it isn't, the window will be opened by the script.

```
if isopen(window Customer_Maintenance) then
  warning "The Customer Maintenance window is already open.";
else
  open form 'Customer Maintenance';
end if;
```

itemdata()

Description

The **itemdata()** function returns the numeric value associated with an item occupying a specified position in a list field: a list box, multi-select list box, combo box, drop-down list, button drop list or visual switch.

Syntax

itemdata(*window_field*, *position***)**

Parameters

- *window_field* – A list field displayed in the window.
- *position* – An integer indicating the numeric position of the item in the list field. The first position in the list has the integer value 1, and so on.

Return value

Long integer

Comments

Numeric values can be associated only with items added to a list field using the **add item** statement. If the item associated with the *position* specified was defined as a static item in the Data Type Definition window, or if no data was specified when the item was added using the **add item** statement, 0 will be returned.

Examples

The following example displays the percentage rate associated with the third selection (City Tax) in the Tax_Type field.

```
warning "The city tax rate is " + str(itemdata(Tax_Type, 3));
```

The following example retrieves the data item associated with the item currently selected in Tax_Type field.

```
local long item_data;

item_data = itemdata(Tax_Type, Tax_Type);
```

Related items

Commands

add item, **countitems()**, **delete item**, **finddata()**, **insert item**, **itemname()**

Additional information

Chapter 6, "Data Types," in Volume 1 of the Dexterity Programmer's Guide

itemname()

Description

The **itemname()** function returns the text value in a specified position in a menu or list field. List fields include list boxes, multi-select list boxes, combo boxes, drop-down lists, button drop lists and visual switches.

Syntax

itemname(**field** / **menu** *name*, *position***)**

Parameters

- *name* – The name of the menu or list field for which you want to return the name of an item. The field must be a window field.
- *position* – An integer indicating the numeric position of the text value you want to return. For list fields, 1 indicates the first position in the list. For menus, 1 indicates the first user-defined menu item. Grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste, are not counted.

Return value

String

Examples

The following example displays a text value from a list field in a warning message.

```
warning "The first customer is " + itemname(field 'Customer Name',1);
```

The following example retrieves the name of the second item in the Markup menu.

```
local string item_name;

item_name = itemname(menu Markup, 2);
```

Related items

Commands

add item, **countitems()**, **delete item**, **finditem()**, **itemdata()**

Additional information

Chapter 6, "Data Types," in Volume 1 of the Dexterity Programmer's Guide

keynumber()

Description

The **keynumber()** function returns the numeric position of a key defined for a table. This allows you to use the key number instead of the key name to access the records in a table.

Syntax

keynumber(table *table_name*, *string_expression***)**

Parameters

- **table** *table_name* – The table containing the key you want to know the position for.
- *string expression* – The name of the key in the table specified by the *table_name* parameter.

Return value

Integer

Examples

The following example sets a local variable named access_type to the number of the By Name key for the Customer_Master table. The key number is then used to access the table using the **change** statement.

```
local integer access_type;

access_type = keynumber(table Customer_Master, "By Name");
change first table Customer_Master by number access_type;
```

length()

Description

The **length()** function returns the length of the specified string or text value.

Syntax

length([*string_expr* / *text_expr*] **)**

Parameters

- *string_expr* | *text_expr* – The string or text value for which you want to find the length.

Return value

Integer

Examples

The following example sets a local variable named number_of_characters to the length of a string. In this case, number_of_characters will be set to 24.

```
local integer number_of_characters;

number_of_characters = length("Account number not found");
```

The following example verifies that the Account Number string field has an invalid length.

```
if length('Account Number') > 30 then
    warning "Account number is too long.";
end if;
```

The following example sets a local variable named text_size to the length of the User Comments text field in a window:

```
text_size = length ('User Comments');
```

lock

Description

The **lock** statement locks one or more window fields. A user won't be able to modify a locked field until it's unlocked with the **unlock** statement, or until the form or window containing the field is restarted.

Syntax

lock {**field**} *field_name* {, *field_name*, *field_name*...}

Parameters

- **field** – An optional keyword identifying*field_name* as a field name.
- *field_name* – The name of a field to be locked.

Comments

The **lock** statement prevents user input into a field. It operates much the same as the **disable field** statement, making the locked field appear with the same properties as if the field's Editable property had been set to false using the Properties window.

*The **lock** statement can't be used in scrolling windows to selectively lock one or more fields on certain lines. However, it can be used before the **fill window** statement to lock one or more fields on all lines of the scrolling window.*

Examples

The following example locks the Customer ID field.

```
lock field 'Customer ID';
```

The following example locks the Customer ID, Company Name and Contact Name fields.

```
lock field 'Customer ID', 'Company Name', 'Contact Name';
```

Related items

Commands
disable field, **unlock**

lower()

Description

The **lower()** function returns a string in which all the alphabetic characters of a specified string have been converted to lower case.

Syntax

lower(*string_expression***)**

Parameters

- *string_expression* – The string or string field you wish to convert to lower case.

Return value

String

Comments

This function can be used to manipulate a string without regard to the case of the characters in the string.

Examples

The following example evaluates the contents of the Password field without regard to the case of the characters in the string.

```
if lower(Password) = Password of table User_Passwords then
    open window Customer_Maintenance;
end if;
```

Related items

Commands

upper()

max()

Description

The **max()** function returns the number with the greater value from a set of two numeric fields, constants or expressions.

Syntax

max(*num1*, *num2***)**

Parameters

- *num1* – Numeric field, constant or expression you wish to compare to the value in the *num2* parameter.
- *num2* – Numeric field, constant or expression you wish to compare to the value in the *num1* parameter.

Return value

Numeric

Comments

The **max()** function can't be used for date or time values.

Examples

The following examples set a variable to the larger number in the function's parameters.

```
i = max(10, 20);       {i = 20}

i = max(10, -20);      {i = 10}

i = max(Total, Subtotal+20);

i = max(finditem(Customer_List,'John Smith'),15);
```

Related items

Commands

min()

min()

Description

The **min()** function returns the number with the lesser value from a set of two numeric fields, constants or expressions.

Syntax

min(*num1*, *num2***)**

Parameters

- *num1* – Numeric field, constant or expression you wish to compare to the value in the *num2* parameter.
- *num2* – Numeric field, constant or expression you wish to compare to the value in the *num1* parameter.

Return value

Numeric

Comments

The **min()** function can't be used for date or time values.

Examples

The following examples set a variable to the smaller number in the function's parameters.

```
i = min(10, 20);   {i = 10}

i = min(10, -20);  {i = -20}

i = min(Total, Subtotal+20);

i = min(finditem(Customer_Field,'John Smith'),15);
```

Related items

Commands

max()

minute()

Description

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

Syntax

minute(*time***)**

Parameters

- *time* – A time or datetime value.

Return value

An integer between 0 and 59.

Examples

The following example sets the variable minute_of_time to the number of minutes in the Starting Time field.

```
local integer minute_of_time;

minute_of_time = minute('Starting Time');
```

Related items

Commands

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

missing()

Description

The **missing()** function ascertains whether an optional parameter has been supplied to a procedure or function.

Syntax

missing(*parameter***)**

Parameters

- *parameter* – The procedure or function parameter that will be checked.

Return value

A boolean. True indicates the parameter was not supplied. False indicates the parameter value was supplied.

Examples

The following procedure script checks whether the optional description parameter is supplied. If the value is supplied, the value is displayed in the Description field. Otherwise, the value "None" is displayed.

```
in string item_name;
optional in string description;

'Item Name' of window Item_Maintenance of form Item_Maintenance =
➥ item_name;

if not missing(description) then
    'Description' of window Item_Maintenance of form Item_Maintenance
    ➥ = description;
else
    'Description' of window Item_Maintenance of form Item_Maintenance
    ➥ = "None";
end if;
```

Related items

Additional information

Chapter 20, "Procedures," in Volume 2 of the Dexterity Programmer's Guide

mkdate()

Description

The **mkdate()** function creates a date value from three integer values.

Syntax

mkdate(*month, day, year***)**

Parameters

- *month* – An integer between 1 and 12 representing the month.
- *day* – An integer between 1 and 31 representing the day.
- *year* – An integer between 1800 and 9999 representing the year. You must supply all four digits of the year.

Return value

A date or datetime value

Examples

The following example sets a local variable named this_date to a date value of 5/28/1998.

```
local date this_date;

this_date = mkdate(5,28,1998);
```

Related items

Commands

addmonth(), **day()**, **dow()**, **eom()**, **month()**, **setdate()**, **sysdate()**, **year()**

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

A time or datetime value

Examples

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

```
local time this_time;

this_time = mktime(14,25,37);
```

Related items

Commands

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

month()

Description

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

Syntax

month(*date***)**

Parameters

- *date* – A date or datetime value.

Return value

An integer between 1 and 12.

Examples

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;

month_of_year = month(sysdate());
```

Related items

Commands

addmonth(), **day()**, **dow()**, **eom()**, **mkdate()**, **setdate()**, **sysdate()**, **year()**

move field

Description

The **move field** statement changes the position of a field in a window.

Syntax

move field *field_name* **to** *h-position*, *v-position*

Parameters

- *field_name* – The name of the field to be moved.
- *h-position* – The new horizontal position of the upper left corner of the field, measured from the left edge of the window.
- *v-position* – The new vertical position of the upper left corner of the field, measured from the top of the window.

Comments

The positions are expressed in pixels. The coordinates 0, 0 specify the upper left corner of the window. The upper left corner of the field will be moved to the specified coordinates. To leave one of the coordinates unchanged, use -1 as the parameter.

Examples

The following example positions the upper left corner of the OK button at the coordinates 5, 200.

```
move field 'OK Button' to 5, 200;
```

This example moves the left edge of the OK button to 5 pixels from the left edge of the window. The vertical position is unchanged.

```
move field 'OK Button' to 5,-1;
```

Related items

Commands

hide field, **resize field**, **show field**, **Field_GetSize()**

move window

Description

The **move window** statement changes the position of a window or scrolling window.

Syntax

move window *window_name* **to** *h-position*, *v-position*

Parameters

- *window_name* – The name of the window or scrolling window to be moved.
- *h-position* – The new horizontal position of the upper left corner of the window or scrolling window. For windows, this value is measured from the left edge of the screen. For scrolling windows, this value is measured from the left edge of the window that contains the scrolling window.
- *v-position* – The new vertical position of the upper left corner of the window or scrolling window. For windows, this value is measured from the top of the screen. For scrolling windows, this value is measured from the top edge of the window that contains the scrolling window.

Comments

The positions are expressed in pixels. The coordinates 0, 0 specify the upper left corner of the screen or window. If a toolbar is present, the vertical position 0 is located below the bottom of the toolbar.

To leave one of the coordinates unchanged, use -1 as the parameter.

When moving a scrolling window, any fields or drawn objects that directly or indirectly touch the scrolling window will also be moved.

Examples

This example moves the left edge of the Customer_Maintenance window to 50 pixels from the left edge of the screen and 50 pixels from the bottom of the toolbar.

```
move window Customer_Maintenance to 50,50;
```

This example moves the left edge of the Customer_Maintenance window to 5 pixels from the left edge of the screen. The vertical position is unchanged.

```
move window Customer_Maintenance to 5,-1;
```

This example moves the Customer_Entry_Scroll scrolling window to 10 pixels from the left edge and 20 pixels from the top edge of the window that contains it.

```
move window Customer_Entry_Scroll to 10, 20;
```

Related items

Commands

resize window, **Window_GetPosition()**, **Window_GetSize()**, **Window_SetBaseSize()**

naterr()

Description

The **naterr()** function returns the native error code or codes for the last table operation on a specified table. If no table is specified, the **naterr()** function returns the native error code or codes for the last table operation, regardless of which table it was performed on. This function should be used only with c-tree and Pervasive.SQL tables.

Syntax

naterr({**table** *table_name*}**)**

Parameters

- **table** *table_name* – The name of the table you wish to check the native error for.

Return value

Long integer

Comments

When the **err()** function returns an error code, you can use the **naterr()** function to find out the native error that occurred.

Refer to the CTERROR.PDF file included with Dexterity for a list of c-tree native error codes. Refer to the Pervasive.SQL documentation or Pervasive Software's web site for a list of Pervasive.SQL native error codes.

For SQL tables, use the SQLError procedure and the Table_GetSQLErrorText() function to retrieve native error information.

Examples

The following example uses the **naterr()** function after a save operation for a Pervasive.SQL or c-tree table to display any native error for a save operation.

```
local long native_error;
local string error_message;

copy to table Seller_Data;
save table Seller_Data;
if err(table Seller_Data) <> OKAY then
    error_message = "An error occurred on table Seller Data. ";
    {Check the native error.}
    native_error = naterr(table Seller_Data);
    if native_error <> OKAY then
        error_message = error_message + "Native error code: "
        ➥ + str(native_error);
    end if;
    error error_message;
end if;
```

Related items

Commands

check error, err()

new

Description

The **new** statement creates a reference to a new COM object. A reference to the specified object is returned.

Syntax

new *object_reference*()

Parameters

- *object_reference* – A fully-qualified COM object type to create.

Examples

The following example creates a new instance of the Microsoft Excel® application.

```
local Excel.Application app;

{Create a new instance of Excel.}
app = new Excel.Application();
```

Related items

Commands

COM_CreateObject(), COM_GetObject()

Additional information

Objects on page 335 of Volume 2 of the Dexterity Programmer's Guide

old()

Description The **old()** function returns the old value of the current field.

Syntax **old()**

Parameters None

Return value Determined by the type of data stored in the field.

Comments This function should be used only in a field change script to retrieve the value that the field contained before it was changed.

*This function should not be used in a change script run using the **run script** or **run script delayed** statements. Focus must be in the field for which this function is run for it to operate properly.*

We don't recommend using the **old()** function with composite fields.

Examples The following example subtracts the old value of a field named Total from its current value to indicate a net change.

```
Total = Total - old();
```

open form

Description

The **open form** statement opens a specified form.

Syntax

open form *form_name* {**return to** *field_name* {, **nofocus**}}

Parameters

- *form_name* – The name of the form to be opened.
- **return to** *field_name* – Indicates the script that's opening the form requires a piece of information to be returned to the specified field in the window from which the form was opened.
- **nofocus** – Including this keyword prevents the focus from moving to the field to which a value is being returned. The change script for the field to which a value is being returned will still be run when the value is returned.

Comments

A typical use for the **open form** statement is in a change script for a button that opens a form. The optional **return to** *field_name* clause signifies that when the form being opened is eventually closed, some piece of information from that form will be returned to the field named in the *field_name* parameter. The post script of the form being opened by this statement should contain a **return** statement that returns the information requested. This feature is useful for lookup windows and creates the basis for inter-form communication.

Examples

The following example opens the Customer_Lookup form; a Customer ID will be returned to the Customer ID field of the Customer_Maintenance window when the Customer_Lookup form closes.

```
open form Customer_Lookup return to 'Customer ID' of window
➥ Customer_Maintenance;
```

The following script is the form post script for the Customer_Lookup form. It returns the Customer ID selected in the Customer_Lookup_Scroll scrolling window.

```
return 'Customer ID' of window Customer_Lookup_Scroll;
```

Related items

Commands

close form, **return**

open form with name

Description

When the application is running, the **open form with name** statement opens the form that has the name specified.

Syntax

open form with name *form_name* {**in dictionary** *product_ID*}

Parameters

- *form_name* – A string containing the name of the form to be opened.
- **in dictionary** *product_ID* – A clause containing the integer product ID for a third-party dictionary.

Comments

The **open form with name** statement is useful for opening forms when the name of the form isn't known when the application dictionary is being created. An example of this is opening a form from a third-party dictionary created by another developer.

Examples

The following example opens a form called Customer Contacts.

```
open form with name "Customer Contacts";
```

The following example opens a form called Billing Setup in the dictionary whose ID is 100.

```
open form with name "Billing Setup" in dictionary 100;
```

open table

Description

The **open table** statement opens a table buffer with the same name as the table, or opens a table buffer for a named table using a different name for the buffer.

Syntax

open table *table_name* {**with name** *string_expr* } {, **exclusive**} {**blocksize** *size*} {**as** *database_type* {, *pathname*}}

Parameters

- *table_name* – The name of the table to be opened.
- **with name** *string_expr* – This clause can be used to open a table buffer for a given table, while naming the newly-opened buffer something other than the table's name. This is used for local anonymous tables.
- **exclusive** – This keyword indicates that the table will be opened for exclusive use, ensuring that only the script or form opening the table will have access to it. Exclusive use of a table is necessary to use the **delete table** statement to delete the table from the operating system or database.
- **blocksize** *size* – This clause only applies when a SQL database type is used. It is an integer value indicating the size of the cursor block to use for actions on this table. The maximum block size allowed is 255 records. If a number greater than 255 is specified, 255 will be used.
- **as** *database_type* – A constant indicating the database type that will be used when opening the table. If this parameter isn't used, the database type specified in the table's definition will be used. The following table lists the constants and corresponding database type:

Constant	Description
DB_TYPE_SQL	Uses SQL as the database type
DB_TYPE_CTREE	Uses c-tree Plus as the database type
DB_TYPE_BTRIEVE	Uses Pervasive.SQL as the database type
DB_TYPE_MEMORY	Uses a memory-based table
DB_TYPE_TABLEDEFAULT	Uses the database type specified in the table definition. If Default is selected in the table definition, the database type specified in the command line or the defaults file **DatabaseType** setting is used.

*Inconsistent use of this parameter can lead to duplicate tables in your application. For example, if a table is created using the SQL database type and later an **open table** statement that includes the **as** DB_TYPE_CTREE option attempts to open the same table, a new c-tree version of that table will be created.*

- *pathname* – A variable string value indicating the location where the table will be opened. This parameter will override the path returned by the Pathname or SQLPath procedure. While the **as** *database_type* parameter can be used alone, the *pathname* parameter can be used only in conjunction with the **as** *database_type* parameter.

  Generic pathnames use a colon (:) before and after server names or drive letters. Forward slashes are used after every directory, folder name or database name.

Comments

You shouldn't have to use this statement very often, as tables are automatically opened when they are accessed by a form.

Examples

The following example opens the table Inventory_Master for exclusive use so it can be deleted using the **delete table** statement.

```
open table Inventory_Master, exclusive;
delete table Inventory_Master;
```

The following example opens the table Inventory_Master in the INVENT database located on the SQL server named Server12.

```
open table Inventory_Master as DB_TYPE_SQL, ":Server12:INVENT/";
```

The following example counts the records in a named table. A table name is passed into the procedure as a string. A local anonymous table buffer named working_table is declared. The table named in the string in parameter is opened and will use the working_table table buffer declared in this script. The **countrecords()** function counts the records in the table.

```
in string table_name;

local anonymous table working_table;
local long record_count;

{Open the table named in the table_name in parameter.}
open table working_table with name table_name;

{Counts the records in the table.}
record_count = countrecords(table working_table);
```

Related items

Commands

copy from table to table, **delete table**

Additional information

Chapter 17, "Memory-based tables," and Chapter 20, "Procedures," in Volume 2 of the Dexterity Programmer's Guide

open window

Description

The **open window** statement opens a specified window.

Syntax

open window *window_name* {**as modal**}

Parameters

- *window_name* – The name of the window to be opened.
- **as modal** – Including this clause causes the window to be opened as a modal window, regardless of how the WindowType property is set for the window.

Comments

This statement is needed to open windows for which the AutoOpen property is set to false. A window with the AutoOpen property set to true is automatically opened when the form it belongs to is opened.

Examples

The following example opens the Customer_History window.

```
open window Customer_History;
```

Related items

Commands

close window

override component

Description

The **override component** statement is used only with extended composite fields. The statement sets the keyable length of a component in an extended composite. It also specifies the character whose width will be used as the base width for the component.

Syntax

override component *component* **of field** *composite_field*, *component_length*, *width_base_character*

Parameters

- *component* – An integer specifying the component to format.
- **of field** *composite_field* – The name of the extended composite field that's being formatted.
- *component_length* – An integer specifying the component's keyable length.
- *width_base_character* – A single-character string whose width will be used as the base width when displaying the component.

Comments

The **override component** statement changes the formatting for a composite component throughout the application. To do this, the statement should be in the form pre script for the Main Menu form of the main dictionary. If the statement is used in an integrating dictionary, it should be executed before opening any forms with windows that display the composite.

The *width_base_character* parameter ensures the component has enough room to be displayed. For example, if the component will be displaying only numbers, 9 can be used as the width-base character. If text will be displayed in the component, W can be used since it's the widest letter.

Examples

The following statement, located in the form pre script for the Main Menu form, overrides the format for the third component of the composite field Customer ID. The component will have a keyable length of 5 and use W for the width base character.

```
override component(3) of field 'Customer ID' of window 'Main Window',
➥ 5, "W";
```

Related items

Commands

override field

Additional information

Chapter 9, "Composites," in Volume 2 of the Dexterity Programmer's Guide

override field

Description

The **override field** statement is used only with extended composite fields. The **override field** statement sets the separator character and specifies whether scroll arrows will appear for the composite field.

Syntax

override field *composite_field*, *separator*, *scroll_arrows*

Parameters

- *composite_field* – The name of the extended composite field being formatted.
- *separator* – A string specifying the separator character to use between components of the composite. The same separator character will appear between all components of the composite.
- *scroll_arrows* – A boolean specifying whether scroll arrows will be displayed for the composite. A true value will display scroll arrows; a false value won't display scroll arrows.

Comments

The **override field** statement changes the formatting for a composite field throughout the application. To do this, the statement must be in the form pre script for the Main Menu form of the main dictionary. If the statement is used in an integrating dictionary, it should be executed before opening any forms with windows containing the composite.

Examples

The following statement, located in the form pre script for the Main Menu form, overrides the format for the composite field Customer ID. A colon (:) will be used as the separator character, and scroll arrows won't be displayed.

```
override field 'Customer ID',":", false;
```

Related items

Commands

override component

Additional information

Chapter 9, "Composites," in Volume 2 of the Dexterity Programmer's Guide

pad()

Description

The **pad()** function adds a specified string to the beginning, end or both the beginning and end of another specified string. It also allows you to set the length of its returned value.

Syntax

pad(*string*, *direction*, *addstring*, *length***)**

Parameters

- *string* – A string field or string value to which the string specified in the *addstring* parameter will be added.
- *direction* – A Dexterity constant that defines where the specified *addstring* should be added to the specified *string*. The possible values for the parameter are:

Constant	Description
LEADING	Add *addstring* to the start of *string*.
TRAILING	Add *addstring* to the end of *string*.
LEADING+TRAILING	Add *addstring* to the beginning and end of *string*.

- *addstring* – A string field or string value to be added to the specified *string* parameter.
- *length* – An integer specifying the length of the string this function returns.

Return value

String

Comments

The *length* parameter specifies the exact size of the returned string. If the resulting concatenation of *string* and *addstring* is shorter than the specified *length*, the *addstring* parameter will be added repeatedly until the *length* is reached. The last addition of *addstring* will be truncated at the point where the specified *length* is reached.

If the resulting concatenation of *string* and *addstring* is longer than the specified *length*, *addstring* will be truncated to the necessary length.

If *addstring* is added to both the beginning and end of the existing string, and an odd number of characters is needed to attain the specified *length*, then the extra character will be added to the end of the string.

Examples

The following example saves the part number and description to a comma-delimited text file.

```
local integer file_ID;
local long result;
local boolean l_boolean;

{open the text file.}
file_ID = TextFile_Open("INVENTORY.TXT", 1, 0);

{Read the items in the table and write them to the text file.}
get first table Inventory_Data;
while err()<>EOF do
    result = TextFile_Write(table_ID, pad('Part Number' of
    ➥ table Inventory_Data, LEADING, " ", 6) + "," +
    ➥ pad('Description' of table Inventory_Data, LEADING, " ", 32));
    get next table Inventory_Data;
end while;
{Close the text file.}
l_boolean = TextFile_Close(table_ID);
```

Related items

Commands

trim()

physicalname()

Description

The **physicalname()** function returns the physical name of a global field, the physical name of a table, or the column name of a field in a SQL table.

Syntax

physicalname(*resource***)**

Parameters

- *resource* – The resource whose physical name is being retrieved.

Comments

The **physicalname()** function is useful when you are using pass-through SQL in your application. The names of tables and columns can be retrieved and used in the pass-through SQL without requiring that you hard-code them.

Examples

The following example retrieves the physical name for the Monthly Utility Cost field.

```
local string resource_name;

resource_name = physicalname('Monthly Utility Cost');
```

The following example retrieves the physical name of the House_Data table.

```
local string resource_name;

resource_name = physicalname(table House_Data);
```

The following example retrieves the column name of the Monthly Utility Cost field in the House_Data table.

```
local string resource_name;

resource_name = physicalname('Monthly Utility Cost' of table
➥ House_Data);
```

Related items

Commands
technicalname()

pos()

Description

The **pos()** function returns a numeric value indicating the starting position of a string within a string or text field, or another string. For example, if the first character of the string you are searching for is found in the fifth position in the string you are searching (the target string), the return value is 5. If the string you are searching for isn't found in the target string, the return value is 0.

Syntax

pos(*target*, *search*, *start***)**

Parameters

- *target* – The string, text field or string field you want to search in.
- *search* – The string you want to search for.
- *start* – The position at which the search will start in the target string.

Return value

Integer

Comments

The **pos()** function is most commonly used in conjunction with **substring()** function. The **pos()** function supplies an in parameter used by the **substring()** function indicating the starting position of the desired substring within the string.

Examples

The following examples return the starting position of a string. In the first example, the string being searched for starts in the third position of the target string, so the return value will be 3.

```
i = pos("abcdef", "cde", 1);
```

In the following example, the string being searched for is not in the target string, so the return value will be 0.

```
i = pos("abcd", "cde", 1);
```

In the following example the string "cde" is found at both the second and seventh positions of the target string; however, with the *start* parameter set to 3, the search begins at the third position of the string, so only the second occurrence of the string is found, and the return value is 7.

```
i = pos("bcdeabcdef", "cde", 3);
```

The following example returns the starting position of the *search* string "test" within the *target* field User Comments. The position within User Comments where the "test" string matched will be returned to the *l_position* local variable.

```
local integer l_position;

l_position = pos('User Comments', "test", 1);
```

The following example uses the **pos()** function in conjunction with the **substring()** function to extract a customer's first name from the Name field in which the customer's name appears in the format last name, first name.

```
local integer l_position;
local string l_fullname, l_firstname;

l_fullname = Name;
{Specify the position of the comma.}
l_position = pos(l_name, ",", 1);
{Retrieve the first name substring from the full name. Start at
l_position +1, to account for the space after the comma in the
l_fullname field.}
l_firstname = substring(l_fullname, l_position +1, length(l_fullname)
➥ - (l_position +1));
```

Related items

Commands

substring()

precision()

Description

Returns the total number of digits available for a variable currency field or variable.

Syntax

precision(*vcurrency***)**

Parameters

- *vcurrency* – A variable currency field or variable.

Return value

An integer containing the number of digits.

Comments

For variable currency fields, the precision value returned is based on the storage size of the underlying data type. The value returned is computed according to this formula:

(2 x Storage Size) - 1

For variable currency variables, the value returned is always 23. For currency fields or variables, the value returned is always 19.

Examples

The following example retrieves the precision of the Interest variable currency field.

```
local integer precision_value;

precision_value = precision(Interest);
```

Related items

Commands
scale()

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; 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 upon 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.*

You can use the ***Table_IsRangeSet()*** *function to find out whether a range is set for a table.*

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

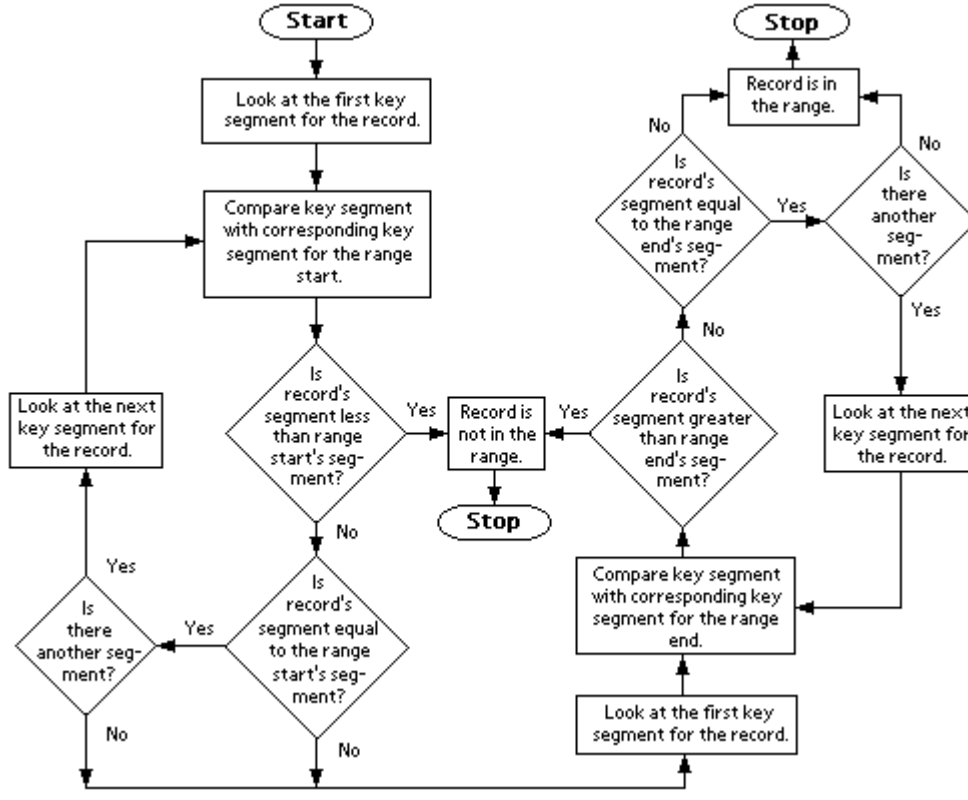
If a key is composed of several segments, you can create ranges based upon several key segments. The **clear field** and **fill** statements are often used when setting ranges for multisegment keys. Typically, the first several corresponding key segments of the range start and range end are set to the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

Evaluating the range

How a range is evaluated depends upon 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.

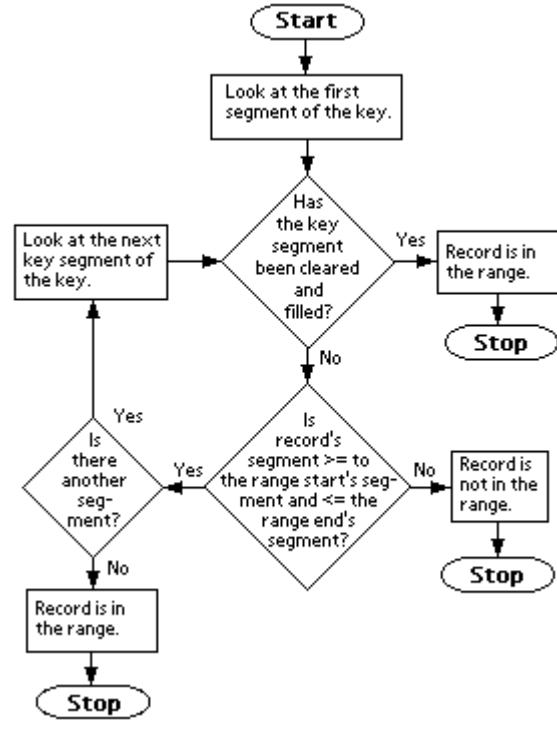
For the Pervasive.SQL and c-tree database managers, ranges are always evaluated *inclusively.* Including the **inclusive** or **exclusive** keyword has no effect on the range.

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



For the SQL database manager, how the range is evaluated depends upon 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 or Pervasive.SQL. 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 upon 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.

Logging range results

You can use the **FHCheckRanges** defaults file setting to log the instances when Dexterity decides which type of range to use. The log will list all instances when Dexterity detected a range that was not "well-behaved" and whether Dexterity used an exclusive or inclusive range. The log will *not* include ranges where the **inclusive** or **exclusive** option was used in the **range end** statement.

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**, **change** or **edit table** statement (one that doesn't include the **prev**, **next**, **first** or **last** keywords), you will receive an EOF error. However, depending upon what value you set the key field to, you could successfully retrieve data by issuing a **get**, **change**, or **edit table** 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 last name between “Doe” and “Smith” will be accessed. 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 Customer_Master;
'Last Name' of table 'Customer_Master' = "Doe";
range start table Customer_Master;
'Last Name' of table 'Customer_Master' = "Smith";
range end table Customer_Master;
```

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.}
'Purchase Date' of table Purchase_Data = 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.}
'Purchase Date' of table Purchase_Data = 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.}
'Purchase Date' of table Purchase_Data = setdate('Purchase Date' of
➥ table Purchase_Data, 11, 17, 1998);
'Store ID' of table Purchase_Data = "A";
range start table Purchase_Data by number 1;
{Set the end of the range.}
'Purchase Date' of table Purchase_Data = setdate('Purchase Date' of
➥ table Purchase_Data, 11, 19, 1998);
'Store ID' of table Purchase_Data = "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**, **sum range**, **Table_IsRangeSet()**

Additional information

Chapter 16, "Ranges," in Volume 2 of the Dexterity Programmer's Guide

range copy

Description

The **range copy** statement copies records in the current range from a source table to a destination table. Only fields common to both tables will be copied. Common fields are those that have the same resource ID. The **range copy** statement will use an exclusive range.

Syntax

range copy table *source_table* **to table** *dest_table* { **by** *key_name* | **by number** *expr* }

Parameters

- **table** *source_table* – The table from which records will be copied.
- **table** *dest_table* – The table to which records will be copied.
- **by** *key_name* | **by number** *expr* – Identifies the key for which the range has been set up on the source table. Keys can be identified by their name (**by** *key_name*) or by their numeric position in the table definition (**by number** *expr*). If no key is specified, the first key of the source table is used.

Comments

If both the source and destination tables are SQL tables, the range copy statement will run as a single SQL statement on the server, greatly improving performance. Otherwise, records in the range will be copied individually.

If a database trigger is registered for either the source or destination table, records will be copied individually.

Examples

The following example creates a range for the Customer_Master table. Records in the current range are copied to the Customer_List table, which is a temporary table used for a report.

```
{Clear any existing ranges associated with this table.}
range clear table Customer_Master;
'Last Name' of table 'Customer_Master' = "Doe";
range start table Customer_Master by number 1;
'Last Name' of table 'Customer_Master' = "Smith";
range end table Customer_Master by number 1;

{Copy the range of records.}
range copy table 'Customer_Master' to table 'Customer_List' by
➥ number 1;
run report 'Customer List';
```

range where

Description

The **range where** statement allows you to apply additional restrictions to a range of records in a table. This statement is available only for SQL tables.

Syntax

range table *table_name* **where** *selection_criteria*

Parameters

- **table** *table_name* – The table to which additional selection criteria is being applied.
- *selection_criteria* – A string containing the additional restrictions to apply to the current range. This string should use syntax similar to a SQL "WHERE" clause. Selection criteria can be applied to any column in the table. You may want to use the **physicalname()** function to retrieve physical names for table columns used in the selection criteria, rather than hard-coding them into the script.

Comments

In Dexterity-based applications, it's common to read a range of records and discard the records that aren't needed. For example, in scrolling windows, the **reject record** statement is used to throw away records that shouldn't be displayed. Because read operations are fast for c-tree and Pervasive.SQL, this is an acceptible design approach. Read operations for SQL Server are much slower, so this approach is inefficient. The **range where** statement can be used to apply additional selection criteria to a range, further narrowing the range of records selected.

The selection criteria added by the **range where** statement is in addition to any other range criteria applied by the **range** statement. To remove the additional selection criteria for the range, issue the **range where** statement with the empty string ("") as the selection criteria. The **range clear** statement will also remove the additional selection criteria.

Because this statement is available only for SQL tables, you will need to have two sets of code. One set will support SQL tables, and the other will support c-tree and Pervasive.SQL tables.

Examples

The following example applies additonal criteria to the Buyer_Data table, used to populate a scrolling window. Only buyers that have a maximum price of less than $80,000 will be displayed. Rather than use the **reject record** statement in the scrolling window fill script to reject records that don't meet the criteria, an additional selection criteria is applied with the **range where** statement.

```
range clear table Buyer_Data;
range table Buyer_Data where physicalname('Maximum Price' of table
➥ Buyer_Data) + "< 80000";
fill window Buyer_Lookup_Scroll;
range clear table Buyer_Data;
```

Related items

Commands
assign as key

redraw

Description

The **redraw** statement updates and redisplays one or more window fields.

Syntax

redraw {**field**} *field_name* {, *field_name*, *field_name*...}

Parameters

- **field –** An optional keyword identifying *field_name* as a field.
- *field_name* – The name of the field to be redrawn. Multiple fields can be redrawn using one **redraw** statement by listing each field's name, separated by commas.

Comments

For performance reasons, list items such as list boxes and combo boxes aren't automatically redrawn when new strings are added to the list with the **add item** statement. All additions to list fields made while a window is open should be followed by a **redraw** statement.

Examples

This example adds an item to the Shipping Method list box and then redraws the field.

```
add item "UPS" to field 'Shipping Method';
redraw field 'Shipping Method';
```

This example adds items to the Shipping Method and Payment Method list boxes and then redraws the fields.

```
add item "UPS" to field 'Shipping Method';
add item "Credit Memo" to field 'Payment Method';
redraw field 'Shipping Method', 'Payment Method';
```

reject record

Description

The **reject record** statement is used in the fill script for a scrolling window. It's used to reject records that you don't want displayed when a scrolling window is filled.

Syntax

reject record

Parameters

None

Comments

Rejecting a record keeps the record from being displayed in a scrolling window only. It doesn't affect the data in the table.

Scrolling performance can be severely reduced if many of the records are rejected, since the scrolling window will continue searching for records until the window is filled. Once the **reject record** statement is encountered in the fill script, the record will be rejected and the fill script will end.

This statement is used in the Security procedure to deny access to a form, report or table. It is also used in trigger scripts to abort processing.

*When a **reject record** statement is executed, the current script is halted, and the remaining scripts in the call stack are aborted.*

When used in the trigger processing procedure for a scrolling window fill script trigger, the reject record statement will stop all pending script operations and prevent the current record from being displayed in the scrolling window.

Examples

In the following example, the fill script for a scrolling window will reject any records that have an account type not equal to 1.

```
if 'Account Type' of table Account_Master <> 1 then
    reject record;
end if;
```

Related items

Additional information

Chapter 10, "Scrolling Windows," in Volume 2 of the Dexterity Programmer's Guide

reject script

Description

The **reject script** statement is used in trigger processing procedures for focus triggers to stop the current script and any pending scripts to be run.

Syntax

reject script

Parameters

None

Comments

Previously, the **reject record** statement was used to stop processing in trigger processing procedures. We recommend that you change your code to use the **reject script** statement instead.

Examples

In the following trigger processing procedure, the value of the Contact Date field is examined. If the field doesn't contain a value, processing is stopped.

```
if empty('Contact Date' of window IG_Contact_History of form
➥ IG_Contact_History) then
    warning "You must supply a contact date.";
    reject script;
end if;
```

Related items

Additional information

Using reject script, in Chapter 9, "Focus Triggers," of the Integration Guide

release table

Description

The **release table** statement releases a record reserved in a table with the **edit table** statement or releases a lock on a record read with the **change** or **edit table** 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.

Examples

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

```
release table Customer_Master;
```

Related items

Commands

change, **edit table**

Additional information

Chapter 15, "Working with Records," and Chapter 18, "Multiuser processing," in Volume 2 of the Dexterity Programmer's Guide

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 either the **edit table** statement or 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 table Customer_Master.

```
remove table Customer_Master;
```

The following example removes records for all the customers from Anderson to Johnson for the table Customer_Master. The range of customers to remove is set first, then the records are removed.

```
'Last Name' of table Customer_Master = "Anderson";
range start table Customer_Master;
'Last Name' of table Customer_Master = "Johnson";
range end table Customer_Master;
remove range table Customer_Master;
```

Related items

Commands

change, **edit table**, **range**

Additional information

Chapter 15, "Working with Records," and Chapter 18, "Multiuser processing," in Volume 2 of the Dexterity Programmer's Guide

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.
```

Examples

The following example will read all the records in a temporary table. Records will be read until an End of File (EOF) error is returned.

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

Related items

Commands

continue, **exit**, **for do...end for**, **while do...end while**

replace()

Description

The **replace()** function allows you to overwrite characters in a string by replacing existing characters in the string.

Syntax

replace(*target*, *replacement*, *start*, *length***)**

Parameters

- *target* – A string or string field containing the characters you wish to replace.
- *replacement* – A string you wish to replace characters in *target* with.
- *start* – The position in the target string where you want to put the replacement characters. Spaces are included in this length.
- *length* – The number of characters from the *replacement* string you want to replace characters in the *target* string with. Spaces are included in this length.

Return value

String

Examples

The following example sets a local variable named new_string to the return value of the **replace()** function. The function replaces characters in the target string "Enter vendor number" with a six-character replacement string of "client". The replacement begins at the seventh character in the target string. As a result, new_string will be set to "Enter client number".

```
new_string = replace("Enter vendor number", "client", 7, 6);
```

The following example replaces characters in the target string "Quarterly sales" with 12 characters from the replacement string "sales report". The replacement begins at the eleventh character in the target string. Since the result extends past the length of the target string, the result is longer than the original target string. In this example, the value returned to new_string will be "Quarterly sales report".

```
new_string = replace("Quarterly sales", "sales report", 11, 12);
```

required()

Description

The **required()** function returns a value indicating whether all the fields in the specified form or window that have the Required property set to true have data in them. If any of the required fields are blank, the return value will be false.

Syntax

required([**form** | **window**] *name***)**

Parameters

- **form** | **window** – The type of resource you're checking.
- *name* – The name of the form or window you're checking.

Return value

Boolean

Comments

A common use of this function is to ensure that all necessary information has been entered in a window when a user attempts to save a record.

Fields currently locked, hidden or disabled aren't checked by the **required()** function, even if they've been marked as required, since fields that are locked, hidden or disabled can't be changed by a user.

Examples

The following example ensures that all necessary information is present in a window before saving a record. If all the required fields have data in them, the return value will be true and the record will be saved. If any of the fields is empty, the return value will be false and a message to that effect will be displayed to the user.

```
if required(window Customer_Maintenance) then
    save table Customer_Master;
else
    warning "Not all required fields have been entered.";
end if;
```

resize field

Description

The **resize field** statement changes the size of a window field.

Syntax

resize field *field_name* **to** *h-size*, *v-size*

Parameters

- *field_name* – The name of the field to be resized.
- *h-size* – An integer specifying the new horizontal width of the field, measured in pixels.
- *v-size* – An integer specifying the new vertical height of the field, measured in pixels.

Comments

This statement doesn't change the top, left position of the field, only the overall horizontal or vertical size. If one of the dimensions is to remain unchanged, use -1 as the parameter.

Examples

In the following example, the Customer ID field is resized to 58 pixels wide and 35 pixels tall.

```
resize field 'Customer ID' to 58, 35;
```

In the following example, the Customer ID field is resized to 58 pixels wide, while the height remains unchanged.

```
resize field 'Customer ID' to 58, -1;
```

Related items

Commands

move field, **Field_GetSize()**

resize window

Description

The **resize window** statement changes the size of a window.

Syntax

resize window *name* **to** *h-size*, *v-size*

Parameters

- *name* – The name of the window to be resized.
- *h-size* – An integer representing the new horizontal width of the window, measured in pixels.
- *v-size* – An integer representing the new vertical height of the window, measured in pixels.

Comments

This statement doesn't change the top, left position of the window, only the overall horizontal or vertical size. If one of the dimensions is to remain unchanged, use -1 as the parameter. If the window size is reduced, fields located outside the new size may no longer be accessible.

The **resize window** statement will resize the window, regardless of how the Resizeable property for the window is set. The **resize window** statement will also resize the controls in the window. To resize the window without resizing controls, use the **Window_SetBaseSize()** function.

Examples

In the following example, the Customer Maintenance window is resized to 500 pixels wide and 315 pixels high.

```
resize window 'Customer Maintenance' to 500, 315;
```

In the following example, the Customer Maintenance window is resized to 250 pixels wide, while the height is unchanged.

```
resize window 'Customer Maintenance' to 250, -1;
```

Related items

Commands

move window, **Window_SetBaseSize()**

Additional information

Resizing windows on page 146 of Volume 1 of the Dexterity Programmer's Guide

resourceid()

Description The **resourceid()** function returns the resource ID for the specified resource.

Syntax **resourceid(***resource***)**

Parameters

- resource – The resource whose resource ID is being returned.

Return value An integer containing the resource ID.

Examples The following example uses the **resourceid()** function to retrieve the resource ID of the Command_IG_Sample form.

```
local integer resID;

resID = resourceid(form Command_IG_Sample);
```

restart field

Description

The **restart field** statement begins processing on the current field again.

Syntax

restart field

Parameters

None

Comments

This statement is useful for fields whose values need to be validated before they're accepted. The difference between this command and the **focus** statement is that the **focus** statement can move the focus to other fields, and always runs the post script of the current field and the pre script for the new field. The **restart field** statement in a change script doesn't run the post script and the pre script for the current field.

Examples

The following example shows a change script that evaluates the name entered in the Salesperson Name field. If the name entered doesn't exist in the Salespeople table, the field is restarted until an existing name is entered.

```
'Salesperson Name' of table Salespeople = 'Salesperson Name' of
➥ window Customer_Maintenance;
get table Salespeople;
if err() <> OKAY then
    warning "Salesperson not found.";
    {Leave the focus on the current field if salesperson is invalid.}
    restart field;
end if;
```

Related items

Commands

focus

restart form

Description

The **restart form** statement restarts the current form.

Syntax

restart form

Parameters

None

Comments

The form is typically restarted when a form or window is saved or cleared. The script associated with the OK or Clear button in a form typically contains a **restart form** statement.

Restarting the form includes clearing the windows for the form, setting the focus to the first field in the tab sequence of the main window of the form, returning all fields to the state specified by their properties, and running the window pre script and the pre script for the first field of the main window. The **restart form** statement won't cause the form pre script to be run.

Table buffers used by the form aren't cleared when the **restart form** statement is run. Use the **clear table** statement to clear the buffers.

The contents of a field will be saved when the form is restarted if the SavedOnRestart property is set to true for that field.

Examples

The following example saves the current item in the Customer_Master table and restarts the form.

```
save table Customer_Master;
restart form;
```

Related items

Commands

clear table

restart script

Description

The **restart script** statement restarts the current script after an exception. This statement is valid only in the **catch** or **else** blocks of the **try...end try** statement.

Syntax

restart script

Parameters

None

Comments

The **restart script** statement is used in structured exception handling to re-execute statements after an exception has occurred. It simply begins executing sanScript statements at the beginning of the script. Local variables and parameters passed into the script retain the values they had prior to the **restart script** statement.

Examples

The following example displays a dialog that asks the user to enter the name of the form to open. If the name is not valid, the form won't be open and an exception will be thrown. Then the script is restarted, allowing the user to enter another name.

```
local string form_name;

try
    if getstring ("Enter the form to open", false, form_name) then
        open form with name form_name;
    else
        exit try;
    end if;
catch [EXCEPTION_CLASS_SCRIPT_MISSING]
    {Catch the exception if the form is missing.}
    error "Cannot find the form " + form_name + ".";
    restart script;
end try;
```

Related items

Commands

restart try, **throw**, **try...end try**

Additional information

Chapter 26, "Structured Exception Handler," in Volume 2 of the Dexterity Programmer's Guide

restart try

Description

The **restart try** statement restarts executing sanScript statements in a **try...end try** block after an exception. This statement is valid only in the **catch** or **else** blocks of the **try...end try** statement.

Syntax

restart try

Parameters

None

Comments

The **restart try** statement is used in structured exception handling to re-execute statements after an exception has occurred. It begins executing sanScript statements at the beginning of the **try...end try** block. Local variables and parameters passed into the script retain the values they had prior to the **restart try** statement.

Examples

The following example attempts to read and actively lock a record from the Seller_Data table. Note that the **throw system exception for table** statement is used to throw a system exception if a database error occurred. If the record can't be actively locked, the script will retry 10 times before displaying a message to the user. The **restart try** statement causes the statements in the **try...end try** block to be executed again, but the value of local variables is not affected.

```
local integer read_count = 0;

try
    'Seller ID' of table Seller_Data = 'Seller ID' of window Sellers;
    {Attempt to actively lock the record.}
    change table Sellers, lock;
    increment read_count;
    {If an error occurred, an exception will be thrown.}
    throw system exception for table Seller_Data;

catch [EXCEPTION_CLASS_DB]
    if err() = LOCKED then
        {Retry the read operation 10 times.}
        if read_count < 10 then
            restart try;
        else
            error "Unable to read and actively lock the record.";
        end if;
    end if;
end try;
```

Related items

Commands

restart script, **throw**, **try...end try**

Additional information

Chapter 26, "Structured Exception Handler," in Volume 2 of the Dexterity Programmer's Guide

restart window

Description The **restart window** statement restarts the current window.

Syntax **restart window**

Parameters None

Comments Restarting a window clears the window buffer, returning all fields to the state specified by their properties, runs the window pre script, moves the focus to the first field in the tab sequence of the window, and runs the field pre script.

The contents of a field will be saved when the window is restarted if the SavedOnRestart property is set to true for that field.

Examples The following example restarts the currently active window.

```
restart window;
```

return

Description

The **return** statement returns a value to a field in a different form, which has requested a value.

Syntax

return *expr*

Parameters

- *expr* – The value to be returned to the form that's requesting a value.

Comments

When another form opens the current form using the **open** statement with the **return to** *field_name* clause, a return path is set up, allowing the current form to return a value to a field in the form that opened it. The **return** statement returns a value to the designated field.

The **return** statement is used for basic inter-form communication and is useful for designing lookup windows. The **return** statement is typically used when closing a lookup window, to return the value of the selected field to the specified field in the opening form.

*When the **return** statement is run, the field that is being returned to must become focused for a value to be returned. If a script on the form being returned to has a **focus** or **restart field** statement that prevents the focus from moving to the returned field, a value won't be returned.*

Examples

The following example uses the **return** statement to return a value to a form. The first script, originating from the Customer_Maintenance window, opens the Customer_Lookup form using the "return to" parameter to set up the return path to the Customer Number field.

```
open form Customer_Lookup return to 'Customer Number' of window
➥ Customer_Maintenance;
```

The following statement will return the current value of the Customer Number field on the Customer_Lookup window to the Customer Number field on the Customer_Maintenance window, using the return path specified when the form was opened.

```
return 'Customer Number';
```

Related items

Commands

open form

round()

Description

The **round()** function returns the rounded value of a specified field. The field can be rounded on either side of the decimal separator.

Syntax

round(*field, side, expression* {, *mode*}**)**

Parameters

- *field* – A currency or variable currency field or value to round.
- *side* – A constant indicating which side of the decimal separator to begin rounding on:

Constant	Description
DECIMALRIGHT	Round the currency value beginning the specified number of places to the right of the decimal separator.
DECIMALLEFT	Round the currency value beginning the specified number of places left of the decimal separator.

- *expression* – An integer indicating the number of places to the right or left of the decimal separator the specified field will be rounded to. The allowable range to the right of the decimal separator is 0 to 5 for currency values and 0 to 15 for variable currency values.
- *mode* – An optional integer that indicates how the value will be rounded. It corresponds to one of the following constants:

Constant	Description	Examples*
ROUNDMODE_UP	Always round up.	4.651 –> 4.66 4.655 –> 4.66 4.659 –> 4.66
ROUNDMODE_DOWN	Always round down.	4.651 –> 4.65 4.655 –> 4.65 4.659 –> 4.65
ROUNDMODE_HALF_UP	If the only digit following the digit to be rounded is 5, round up.	4.675 –> 4.68 4.6751 –> 4.68
ROUNDMODE_HALF_DOWN	If the only digit following the digit to be rounded is 5, round down.	4.675 –> 4.67 4.6751 –> 4.68
ROUNDMODE_HALF_EVEN	If the only digit following the digit to be rounded is 5 and the preceding digit is odd, round up. Otherwise, truncate.	4.675 –> 4.68 4.685 –> 4.68

Constant	Description	Examples*
ROUNDMODE_CEILING	Always round toward positive infinity.	4.655 –> 4.66 -4.655 –> -4.65
ROUNDMODE_FLOOR	Always round toward negative infinity.	4.655 –> 4.65 -4.655 –> -4.66

*Rounding to two decimal places

If this parameter isn't supplied, ROUNDMODE_HALF_UP is used.

Return value

Currency or variable currency

Examples

The following example rounds 1050.123 to 1050.12000, or to two places to the right of the decimal separator.

```
a = round(1050.123, DECIMALRIGHT, 2);
```

The following example rounds the Currency Amount field, containing the value 1050.12, to two places to the left of the decimal separator or 1100.00000.

```
a = round('Currency Amount', DECIMALLEFT, 2);
```

The following example rounds 1050.1238 to one decimal digit more than the number specified in the user's operating system's decimal digits setting. The **currencydecimals()** function returns the current value of this operating system setting (in this case, 2). Therefore, 1050.1238 is rounded to 1050.1240, or three decimal places to the right of the decimal separator.

```
a = round(1050.1238, DECIMALRIGHT, currencydecimals() + 1);
```

The following example rounds 1050.123 to 1050.13, or to two places to the right of the decimal separator. The ROUNDMODE_CEILING constant is used to specify the value will be rounded toward positive infinity.

```
a = round(1050.123, DECIMALRIGHT, 2, ROUNDMODE_CEILING);
```

Related items

Commands

currencydecimals(), **set precision of**, **truncate()**

run application

Description

The **run application** statement starts another application.

Syntax

run application *application*, *parameters*

Parameters

- *application* – A string expression or variable containing the name (and path, if necessary) of the application to be run. If the path is included, it must be specified in native format.
- *parameters* – Any parameters to include when starting the application, such as the name of a file to open. If no parameters are required, a comma and pair of empty quotation marks must be used (," ").

Comments

For DOS and Windows applications, you don't need to specify the complete pathname for the application if the path is already stated in the DOS Path command in the AUTOEXEC.BAT file.

The parameters are passed to the application exactly as they are entered in *parameters*. To use these parameters to open a file along with the application on a Macintosh, you must specify the complete pathname of the file to open. For Windows you must specify the complete pathname for the file to open if required by the application.

In Windows, you can open a file simply by passing an empty string to the *application* parameter and a complete path to the file as the second parameter. Be sure the path to the file is in native format. Based on the file's extension, Windows will attempt to launch the appropriate application. For example, launching the file TEST.HTM would launch a web browser.

Examples

The following example uses the **getfile()** function to prompt the user for a name of an application to run. The APPFILE constant is used so only application files will be displayed in the getfile dialog box. The **run application** statement starts the application. Notice that the pathname is converted to native format.

```
local string application_name;

if getfile("Select an application", APPFILE, application_name) then
    application_name = Path_MakeNative(application_name);
    run application application_name, "";
end if;
```

The following example will start Microsoft Excel for Windows and open the Customer List file, CUSTLIST.TXT, a report exported from Report Writer. The complete pathname is specified.

```
run application "C:\EXCEL\EXCEL.EXE", "C:\REPORTS\CUSTLIST.TXT";
```

The following example will start Windows Write without opening a file.

```
run application "C:\WINDOWS\WRITE.EXE", "";
```

The following example starts the application associated with the file BUYERS.HTM, and loads the file into that application.

```
run application "", "C:\REPORTS\BUYERS.HTM";
```

Related items

Commands

getfile()

run command

Description

The **run command** statement performs the action for the specified command.

Syntax

run command *command_name*

Parameters

- *command_name* – The name of the command to be run.

Comments

If the command has been disable with the **disable command** statement, it will not be run. If only the tag for a command is available, you can use the **Command_Execute()** function to perform the command's action.

Examples

The following example runs the CMD_HousesReport command that prints a report.

```
run command CMD_HousesReport;
```

Related items

Additional information
Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

run macro

Description

The **run macro** statement starts a recorded Dexterity macro.

Syntax

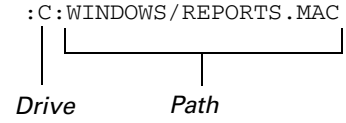
run macro *file_name*

Parameters

- *file_name* – A string expression or variable containing the file name (and path, if necessary) of the macro to be run.

Comments

Pathnames are specified using a generic convention that is designed to be platform independent. The volume or drive name is enclosed in colons (:) followed by the path, using forward slashes (/) to separate the directories or folders. The following example is a valid generic pathname:



You don't need to specify a pathname for the macro if it's already stated by the DOS Path command in the AUTOEXEC.BAT file.

Examples

The following example uses the **getfile()** function to prompt the user for a name of a macro to run. The MACROFILE constant is used so only macro files will be displayed in the getfile dialog box. The **run macro** statement starts the macro.

```
local string file_name;

if getfile ("Select a macro to run.", MACROFILE, file_name) then
     run macro file_name;
end if;
```

Related items

Commands

getfile()

run report

Description

The **run report** statement prepares a report to be printed that's been defined using the Report Writer.

Syntax

run report *report_name* {**with restriction** *boolexp*}{**sort by** *field*{**nocase**}{**descend**}}{**by** *key_name* | **by number** *expr*} {**legends** [*string_expr_list* | *array_index_list*]}{**destination** *screen_boolean*, *printer_boolean* {, *export_type*, *export_name*}}{**printer** *printer_settings*}

Parameters

- *report_name* – The name of a predefined report.

- **with restriction** *boolexp* – Any restrictions you want placed upon the report, in addition to the restrictions defined in the report itself. *Boolexp* is a boolean expression used to define the restriction criteria. The restrictions can be based upon any fields in the report's main table, or any tables related to the main table. If you include two or more restrictions, enclose each restriction in parentheses, and enclose the entire set of restrictions in parentheses as well. You can link the different restriction clauses using **or**, **and** or **not**.

Be sure to fully qualify the field names used in this parameter. If you don't, your script will compile but you will receive an R_tree error when the script runs.

- **sort by** *field* – Allows you to sort the report by the contents of any field in the report, where *field* is the name of the field the sort will be based upon. You can include multiple fields in the **sort by** clause by listing them in their order of precedence, separated by commas. Be sure to fully qualify each field name used in this clause. The sorting specified using this parameter overrides sorting specified in the report definition.

  Two additional sorting options are available to ignore case or sort the report in descending order. These keywords can be applied to each field used in the **sort by** clause.

- **nocase** – If included, sorting won't be affected by whether the field's values are in uppercase or lowercase text. If not included, lowercase text will come before uppercase text in the sorting process.

- **descend** – If included, the report data will be sorted in descending order; from 10 to 1 or Z to A, for example. If not included, the report data will be sorted in ascending order.

- **by** *key_name* | **by number** *expr* – Specifies which table key will be used to access the records in the report's main table. You can identify the key to be used by its name (**by** *key_name*) or by its numeric order in the list of table keys (**by number** *expr*). If this parameter isn't included, the key specified in the report definition will be used.

- **legends** *string_expr_list* | *array_index_list* – Allows you to specify strings to be displayed in legend fields on the report. The strings are separated by commas in the *string_expr_list*. Legend fields typically display information about the range of data included on the report, or how the report is sorted. If there are two or more legend fields on the report, list the strings in the appropriate order: the first value will be displayed in the Legend [1] field on the report, and so on.

  Legend field values can also be used as the array indexes for array fields in calculated fields. Use the *array_index_list* to specify the appropriate numbers to be used as the indexes for the array fields; the values will be converted to long integers in the calculated field expressions and used as the array indexes.

- **destination** *screen_boolean*, *printer_boolean {export_type, export_name}* – Allows you to send the report to the screen, the printer, an export file, or any combination of the three. If this parameter isn't included, when this statement is executed at runtime, the user will have to select a destination from the Report Destination window.

  Set *screen_boolean* to true to send the report output to the screen. Set *printer_boolean* to true to send the report output to the printer.

  Use the *export_type* and *export_name* parameters to send the report output to an export file. The following provides a detailed explanation of each of these parameters:

- *export_type* – A constant that allows you to specify the format for a report sent to an export file. Use one of the following constants:

Constant	Description
TEXTFILE	Text file
TABFILE	Tab-delimited file
COMMAFILE	Comma-delimited file
HTMLFILE	HTML file
PDFFILE	Adobe PDF (Portable Document Format) file

If the data from the report will be appended to an existing text file, the APPENDFILE constant must be included with the export type constant; for instance, `TEXTFILE+APPENDFILE` indicates that the report will be appended to an existing text file. The name of the existing text file should be defined in the *export_name* parameter.

You cannot export a report to a PDF file and print it to a printer at the same time.

- *export_name* – A text string containing the name and path (in generic format) that the export file will be saved with. If you don't supply a path, Dexterity will save the file in the current directory.

- **printer** *printer_settings* – This clause specifies a particular printer to which this report should be printed. The *printer_settings* parameter should be set to the value of the text field that was returned by the **Printer_Define()** function when the printer was defined.

  This clause is independent of the **destination** clause. If this clause is included but the **destination** clause isn't, the Report Destination window will appear when this statement is executed at runtime. If the Printer option is marked in the Report Destination window, the printer specified in this clause will be the printer named in the Print dialog that appears when the Report Destination window is closed.

  If both the **destination** clause and the **printer** clause are included in a **run report** statement, the **destination** clause takes precedence. For example, if the *printer_boolean* parameter is set to false, the report will not be sent to a printer, even if the **printer** clause is used.

Examples

The following example prints the Customer List report, listing only customers from Arizona and sorting them by last name. The legend field will display the range of data printed on the report.

```
run report 'Customer List' with restriction State of table
➥ Customer_Data = "AZ" sort by 'Last Name' legends
➥ "Arizona customers", "By last name";
```

The following example prints the Customer List report, sorted by the State and Company Name fields of the Customer_Data table.

```
run report 'Customer List' sort by State of table  Customer_Data,
➥ 'Company Name' of table Customer_Data;
```

The following example prints the Customer List report, with two restrictions to list only customers from Arizona with preferred status.

```
run report 'Customer List' with restriction ((State of table
➥ Customer_Data = "AZ") and (Status of table Customer_Data =
➥ "Preferred"));
```

The following example saves the Customer List report, sorted by the key L_name_key, as a tab-delimited file with the file name CUSTLIST.TXT. The report won't be displayed or printed to the printer.

```
run report 'Customer List' by L_name_key destination false, false,
➥ TABFILE, "CUSTLIST.TXT";
```

The following example appends the Customer List report to the file CUSTLIST.TXT. The report won't be displayed or printed.

```
run report 'Customer List' destination false, false,
➥ TABFILE+APPENDFILE,"CUSTLIST.TXT";
```

The following example sends the Customer List report to the printer specified in the Printer_Settings field of the Printer_Options table. The **destination** clause is used to ensure that the report prints to the screen as well as the printer.

```
run report 'Customer List' destination true, true, printer
➥ Printer_Settings of table Printer_Options;
```

Related items

Commands

Printer_Define(), **Printer_GetName()**, **run report with name**

Additional information

Chapter 34, "Restrictions," Chapter 37, "Legends," and Chapter 39, "Using Reports in Applications," in Volume 1 of the Dexterity Programmer's Guide
The Printer function library in the Function Library Reference manual

run report with name

Description

The **run report with name** statement prepares a report to be printed. The report to print is specified at runtime.

Syntax

run report with name *report_name* {**with restriction** *boolexp*}{**sort by** *field_name*{**nocase**}{**descend**}}{**by** *key_name* | **by number** *expr*} {**legends** [*string_expr_list* | *array_index_list*]}{**destination** *screen_boolean*, *printer_boolean* {, *export_type*, *export_name*}}{**printer** *printer_settings*} {**in dictionary** *product_ID*}

Parameters

- *report_name* – A string variable or expression containing the name of a predefined report.

- **with restriction** *boolexp* – Any restrictions you want to place upon the report, in addition to the restrictions defined in the report itself. *Boolexp* is a boolean expression used to define the restriction criteria. The restrictions can be based upon any fields in the report's main table, or any tables related to the main table. If you include two or more restrictions, enclose each restriction in parentheses, and enclose the entire set of restrictions in parentheses as well. You can link the different restriction clauses using **or**, **and** or **not**.

Since the report to be run won't be specified until runtime, the compiler will not be able to catch restriction errors, as it will not be able to verify whether fields used in the boolexp *actually exist in the report's tables. **All** table fields used in the restriction **must** be part of tables attached to the report. Also, be sure to fully qualify the field names used in this parameter. If you don't, your script will compile but you will receive an R_tree error when the script runs.*

- **sort by** *field_name* – Allows you to sort the report by the contents of any field in the report, where *field_name* is the name of the field the sort will be based upon. You can include multiple fields in the **sort by** clause by listing them in their order of precedence, separated by commas. Be sure to fully qualify each field name used in this parameter. The sorting specified using this parameter overrides sorting specified in the report definition.

Since the report to be run won't be specified until runtime, the compiler will not be able to catch any sorting errors, as it will not be able to verify whether fields specified in this parameter are actually used in the report.

Two additional sorting options are available to ignore case or sort the report in descending order. These keywords can be applied to each field used in the **sort by** clause.

- **nocase** – If included, the case of field values won't affect sorting. If not included, lowercase text will come before uppercase text in the sorting process.

- **descend** – If included, the report data will be sorted in descending order. If not included, the report data will be sorted in ascending order.

- **by** *key_name* | **by number** *expr* – Specifies which table key will be used to access the records in the report's main table. You can identify the key to be used by its name (**by** *key_name*) or by its numeric order in the list of table keys (**by number** *expr*). If this parameter isn't included, the key specified in the report definition will be used.

- **legends** *string_expr_list* | *array_index_list* – Allows you to specify strings to be displayed in legend fields on the report. Separate the strings by commas in the *string_expr_list*. Legend fields typically display information about the range of data included on the report, or how the report is sorted. If there are two or more legend fields on the report, list the strings in the appropriate order: the first value will be displayed in the Legend [1] field on the report, and so on.

  Legend field values can also be used as the array index for array fields in calculated fields. Use the *array_index_list* to specify the appropriate numbers to be used as the indexes for the array fields; the values will be converted to long integers in the calculated field expressions and used as the array indexes.

- **destination** *screen_boolean*, *printer_boolean {export_type, export_name}* – Allows you to send the report to the screen, the printer, an export file, or any combination of the three. If this parameter isn't included, when this statement is executed at runtime, the user will have to select a destination from the Report Destination window.

  Set *screen_boolean* to true to send the report output to the screen. Set *printer_boolean* to true to send the report output to the printer.

  Use the *export_type* and *export_name* parameters to send the report output to an export file. The following provides a detailed explanation of each of these parameters:

- *export_type* – A constant that allows you to specify the format for a report sent to an export file. Use one of the following constants:

Constant	Description
TEXTFILE	Text file
TABFILE	Tab-delimited file
COMMAFILE	Comma-delimited file
HTMLFILE	HTML file
PDFFILE	Adobe PDF (Portable Document Format) file

  If the data from the report will be appended to an existing text file, the APPENDFILE constant must be included with the export type constant; for instance, `TEXTFILE+APPENDFILE` indicates that the report will be appended to an existing text file. The name of that text file should be defined in the *export_name* parameter.

  You cannot export a report to a PDF file and print it to a printer at the same time.

- *export_name* – A text string containing the name and path (in generic format) that the export file will be saved with. If you don't supply a path, Dexterity will save the file in the current directory.

- **printer** *printer_settings* – This clause specifies a particular printer to which this report should be printed. The *printer_settings* parameter should be set to the value of the text field that was returned by the **Printer_Define()** function when the printer was defined.

  This clause is independent of the **destination** clause. If this clause is included but the **destination** clause isn't, the Report Destination window will appear when this statement is executed at runtime. If the Printer option is marked in the Report Destination window, the printer specified in this clause will be the printer named in the Print dialog that appears when the Report Destination window is closed.

  If both the **destination** clause and the **printer** clause are included in a **run report with name** statement, the **destination** clause takes precedence. For example, if the *printer_boolean* parameter is set to false, the report will not be sent to a printer, even if the **printer** clause is used.

- **in dictionary** *product_ID* – A clause containing the integer product ID for a third-party dictionary from which you want a report run.

Comments

You may not always be able to use the **with restriction** and **sort by** clauses, since you won't know which report is going to be run. However, these clauses can be useful if you have a limited set of reports that could be run using the **run report with name** statement, and they all have common tables and fields upon which you can set restrictions or sorting options.

Examples

In the following example, Customer Reports is a combo box listing available reports. The currently selected report will be run.

```
run report with name 'Customer Reports';
```

The following example saves the report selected in the Customer Reports combo box as a text file. The name of the file will be the report name with the extension .TXT.

```
run report with name 'Customer Reports' destination false, false,
➥ TEXTFILE, 'Customer Reports'+".TXT";
```

The following script prints the Customer Billing report from the third party dictionary whose ID is 100.

```
run report with name "Customer Billing" in dictionary 100;
```

In the following example, Customer Reports is a combo box listing the reports available. The report currently selected in the field will be run. Since there are only two choices in the Customer Reports combo box, and each choice uses the Customer_Data table as its main table, you can use the **with restriction** and **sort by** clauses of the **run report with name** statement. The restriction uses the information in the state_choice field on the current window.

```
run report with name 'Customer Reports' with restriction State of
➥ table Customer_Data = state_choice sort by Last_Name of table
➥ Customer_Data;
```

Related items

Commands

run report

Additional information

Chapter 34, "Restrictions," Chapter 37, "Legends," and Chapter 39, "Using Reports in Applications," in Volume 1 of the Dexterity Programmer's Guide
The Printer function library in the Function Library Reference manual

run script

Description

The **run script** statement runs the change script for a given field.

Syntax

run script *field_name*

Parameters

- *field_name* – The name of the field whose change script will be run.

Comments

This statement is useful when the value in the field is changed by another script and the change should be acted upon. This statement is also useful if there are several fields that use the same script, as it allows you to place the script on one field and then simply run it from the other fields, rather than re-creating the same script for each field.

The **run script** statement requires an additional call stack. For information about call stacks, refer to Chapter 24, "Call Stacks," in Volume 2 of the Dexterity Programmer's Guide.

*We suggest that you use form procedures in place of field change scripts started with the **run script** statement.*

You can't use the **run script** statement to run a field change script containing the **old()** or **diff()** functions. These functions won't work because the focus isn't on the field whose change script is being run.

If you're integrating with Microsoft Dynamics™ GP, you won't be able to run a field's change script using the **run script** statement, if the script contains the **old()** or **diff()** function. If you're running your own field change script, duplicate the functionality of the script by using a form procedure. Pass the current and previous values of the field as parameters for the form procedure. This will allow you to use the script's functionality from any fields in the dictionary, rather than just one specific field.

Examples

The following example shows a script for a Quantity field. The change script for the Extended Price field will update the total price:

```
'Extended Price' = 'Quantity' * 'Price';
run script 'Extended Price';
```

Related items

Commands

call, **run script delayed**

Additional information

Chapter 24, "Call Stacks," in Volume 2 of the Dexterity Programmer's Guide

run script delayed

Description

The **run script delayed** statement runs the change script for a field after the current script is completed. This can be used to prevent recursive script calls.

Syntax

run script delayed *field_name*

Parameters

- *field_name* – The name of the field whose change script you wish to run.

Comments

A maximum of six scripts started with the **run script** statement can be waiting to be processed. The **run script delayed** statement doesn't have this limit because the current script will finish processing before the next script is started.

Examples

In the following example, the change script for the Print Reports field will be run after the current script is completed.

```
run script delayed 'Print Reports';
```

Related items

Commands

run script

Additional information

Chapter 24, "Call Stacks," in Volume 2 of the Dexterity Programmer's Guide

save table

Description

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

Syntax

save table *table_name*{, **norelease**}

Parameters

- *table_name* – The name of the table that will have its table buffer contents saved.
- **norelease** – An optional keyword that prevents an active lock from being released after the save operation.

Comments

After the **save table** statement is run, any passive or active lock is released, regardless of whether the save operation was successful. You can use the **norelease** keyword to prevent an active lock from being released by the **save table** statement.

After a successful save operation, the **save table** statement automatically updates the fields in the table buffer that were changed by other users. This means that the values in the table buffer after the save operation may differ from the values in the table buffer before the save operation. After a successful save operation, you may want to update the data displayed in your application to reflect these changes.

Examples

The following example copies the contents of the auto-copy fields from the window buffer to the table buffer. The table buffer is saved in the Customer_Master table.

```
copy to table Customer_Master;
save table Customer_Master;
```

Related items

Additional information

Chapter 15, "Working with Records," and Chapter 18, "Multiuser processing," in Volume 2 of the Dexterity Programmer's Guide

savefile()

Description

The **savefile()** function creates a dialog box allowing a user to enter a file name and select a path for the file's location. It returns a boolean value indicating which button the user clicked in the dialog box.

The file types displayed in the dialog can be a predefined set, or you can create your own custom set of file types.

Syntax

savefile(*prompt*, *file_type*, *path* {,*custom_file_types*}**)**

Parameters

- *prompt* – The string to display in the dialog box.
- *file_type* – An integer variable or field that the file type value will be returned to. If you don't supply a custom set of file types, the value returned will correspond to one of the following constants:

Constant	Description
TEXTFILE	Text file
TABFILE	Tab-delimited file
COMMAFILE	Comma-delimited file
HTMLFILE	HTML file

  The value indicates the file type selected by the user. You can then use this value to set up report outputs or export routines according to the file type selected.

  If you create a set of custom file types, the value returned indicates which of the file types the user selected. The value 1 indicates the first custom file type, 2 indicates the second custom file type, and so on.

- *path* – A string field or variable. You can set the value of this variable before calling the **savefile()** function to specify the default filename and location that will appear in the dialog box. If you supply only a filename, the current path will be used. If you supply a complete path (in native format) the default location will also be set in the dialog. If you set this variable to the empty string (""), the default filename will have the form "Export#", where # is an integer value.

  If the user clicks Save in the dialog, the path for the file, including the user-entered name and extension will be returned. If the user clicks Cancel, the empty string ("") will be returned.

- *custom_file_types* – An optional string that specifies a custom set of file types to use. The set of custom file types is defined using the following syntax:

  description |*type*|

  The *description* parameter is a string that describes the file type, such as "Comma-Separated Values (*.CSV)". It must be followed by a pipe (|) character.

  The *type* parameter specifies a file extension, such as "*.CSV". When this file type is used, files that match this extension will be displayed in the dialog box. The *type* parameter should not contain spaces. End the *type* parameter with a pipe (|) character.

  Several custom file types can be concatenated within the *custom_file_types* string.

Return value

A boolean indicating which button was clicked. If Save is clicked, the value true is returned. If Cancel is clicked, the value false is returned.

Comments

If you will be using custom file type, consider creating messages or constants for each file type. You can then easily concatenate the filter types together and build the *custom_file_types* string. This is shown in the third example.

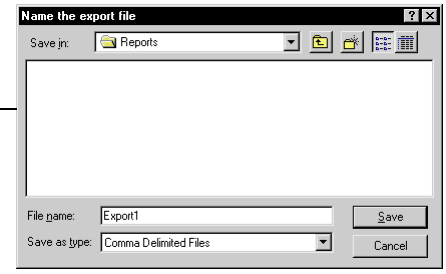
Examples

The following example retrieves a file name, path and export file type using the **savefile()** function. If the user clicks OK and a valid file name has been entered, the output of the Customer List report is saved using the name, location and file type indicated in the dialog box. If the return value is false, the script ascertains whether an error occurred, or the user clicked Cancel.

```
local string file_name;
local boolean did_save;
local integer file_type;

if savefile ("Name the export file", file_type, file_name) then
    run report 'Customer List' destination false, false, file_type,
    ➥ file_name;
else
    did_save = false;
end if;
```

*The **savefile()** function in the previous example would display this dialog box.*



The following example uses a custom file type list so that tab-delimited and comma-separated documents are the only available file types.

```
local string path;
local integer file_type;

if savefile("Export file", file_type, path,
➥ "Tab-delimited (*.txt)|*.txt|Comma-separated (*.csv)|*.csv|") then
    if file_type = 1
        call Export_Tab_File(path);
    else
        call Export_CSV_File(path);
    end if;
end if;
```

The following example shows how constants can be used to create custom file type lists. The following constants have been added to the current dictionary:

TEXT_FILES	**Text (*.txt)|*.txt|**
TAB_FILES	**Tab-delimited (*.txt)|*.txt|**
COMMA_FILES	**Comma-separated (*.csv)|*.csv|**

The following script uses these constants to create a custom file type list for the **savefile()** function:

```
local string path;
local integer file_type;

if savefile("Export file", file_type, path, TEXT_FILES + TAB_FILES +
➥ COMMA_FILES) then
    call Export_File(file_type, path);
end if;
```

scale()

Description

Returns the number of decimal digits for a variable currency field or variable.

Syntax

scale(*vcurrency***)**

Parameters

- *vcurrency* – A variable currency field or variable.

Return value

An integer containing the number of digits.

Comments

For variable currency fields, the scale value returned is based on the decimals setting of the underlying data type. For variable currency variables, the value returned is based on the value stored in the field. For currency fields or variables, the value returned is always 5.

Examples

The following example retrieves the scale of the Interest variable currency field.

```
local integer scale_value;

scale_value = scale(Interest);
```

Related items

Commands
precision()

second()

Description

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

Syntax

second(*time***)**

Parameters

- *time* – A time or datetime value.

Return value

An integer between 0 and 59.

Examples

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;

second_of_time = 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.

*To simplify sanScript code, the **set** statement isn't typically used. Instead, values of fields and variables are set directly using the equals sign (=).*

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 control 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' to "Bob Smith";
set 'Number of Customers' to 156;
```

The following example shows the same assignments made with the equals sign.

```
'Customer Name' = "Bob Smith";
'Number of Customers' = 156;
```

set precision of

Description

The **set precision of** statement sets the number of decimal places in a specified currency or variable currency field to the value of an expression.

Syntax

set precision of {**field**} *field_name* **to** *expression*

Parameters

- **field** – Identifies *field_name* to follow as the name of a field.
- *field_name* – The currency or variable currency field for which the number of decimal places will be set.
- *expression* – An integer indicating the number of decimal places to be used. For currency fields, the allowable range to the right of the decimal separator is 0 to 5. For variable currency fields, it is 0 to 15.

Comments

The **set precision of** statement doesn't change the value of data in a table; it affects only how the data is displayed in a window. Currency values displayed with fewer than the existing number of decimal places will be truncated, *not rounded*.

If a currency field has a linked format, that format will override the **set precision of** statement. Refer to *The Multiple Format Selector* on page 80 in Volume 1 of the Dexterity Programmer's Guide for more information about linked formats.

Examples

The following example shows how the **set precision of** statement controls the number of decimal digits displayed for the Sales Cost field.

```
{'Sales Cost' is stored as 123.9373}
set precision of 'Sales Cost' to 2;
{'Sales Cost' is now displayed as 123.93}
```

The following example increases the number of decimal places displayed for the Sales Cost field.

```
{'Sales Cost' is stored as 125.2}
set precision of 'Sales Cost' to 3;
('Sales Cost' is now displayed as 125.200}
```

The following example sets the number of decimal digits displayed in the Sales Cost field to one more than the user's operating system's decimal digits setting. The **currencydecimals()** function is used to return this current value of this setting.

```
set precision of 'Sales Cost' to 1 + currencydecimals();
```

Related items

Commands
currencydecimals()

set title of window to

Description

The **set title of window to** statement allows you to change the display name of a window at runtime.

Syntax

set title of window *window_name* **to** *string*

Parameters

- *window_name* – The name used to refer to the window in scripts.
- *string* – A string field or string value of up to 78 characters to which the display name of the specified window should be changed. Strings longer than 78 characters will be truncated.

Comments

This statement will not overwrite the window title in the dictionary. If this statement isn't used, the window title specified in the window's Title property will be used.

Examples

The following example uses the **set title of window to** statement in the window's pre script to customize the Customer_Info window's display name for the ABC Construction company.

```
set title of window Customer_Info to "ABC Construction's Customers";
```

The following example uses this statement in a window's pre script to change the window's display name to the name of the company using the application. Since this company name will vary, the information is retrieved from a global variable, owning_company. This variable is set when the company first installs the application.

```
set title of window Customer_Info to owning_company of globals;
```

Related items

Commands
Window_GetMainWindowTitle(), Window_GetTitle()

setdate()

Description

The **setdate()** function creates a date value or modifies 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

A date or datetime value

Examples

The following changes the value of the Start Date field. The *month* parameter is set to 0, so the month value currently displayed in the Start Date field won't be modified. The day is set to the 12th day of the month, and the year is set to 1998.

```
'Start Date' = setdate('Start Date',0,12,1998);
```

Related items

Commands

addmonth(), **day()**, **dow()**, **eom()**, **mkdate()**, **month()**, **sysdate()**, **year()**

show command

Description

The **show command** statement makes the named command visible every place it is displayed, such as a menu or toolbar.

Syntax

show command *command_name* {,*command_name*}

Parameters

- *command_name* – The name of the command to be shown.

Examples

The following example shows the CMD_HousesReport command. Any place the command is displayed, such as a menu or toolbar will be made visible.

```
show command CMD_HousesReport;
```

The following example shows the CMD_Buyers and CMD_Sellers commands.

```
show command CMD_Buyers, command CMD_Sellers;
```

Related items

Commands

hide command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

show field

Description

The **show field** statement makes the specified field or fields visible and accessible to the user.

Syntax

show field *field_name*{, *field_name, field_name*...}

Parameters

- *field_name* – The name of a hidden field to show.

Comments

This statement will display the named field or fields only if they were made inaccessible by the **hide field** statement. The **show field** statement cannot make a field visible whose Visible property is set to false in the Properties window.

Examples

The following example makes the Discount Percent field visible.

```
show field 'Discount Percentage';
```

The following example makes the Discount Percentage and Calculation Method fields visible.

```
show field 'Discount Percentage', 'Calculation Method';
```

Related items

Commands
hide field

show menu

Description

The **show menu** statement shows the specified menu item.

Syntax

show menu *menu_name*, *number*

Parameters

- *menu_name* – The name of the menu containing the item you want shown.
- *number* – An integer representing the placement of the item to be shown in the menu.

Comments

The **show menu** statement cannot be used to show grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste. For more information about menu items that can be hidden and shown, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

Examples

The following example shows the third item in the Transactions menu.

```
show menu Transactions, 3;
```

Related items

Commands

hide menu

show window

Description

The **show window** statement makes the specified scrolling window visible and accessible to the user.

Syntax

show window *window_name*

Parameters

- *window_name* – The name of the scrolling window to show.

Examples

The following example makes the Customer_Entry_Scroll scrolling window visible.

```
show window Customer_Entry_Scroll;
```

Related items

Commands
hide window

str()

Description

The **str()** function returns a string representation of a value that is not a string. This is useful for any situation in which you want to include values that aren't strings in a string expression.

Syntax

str(*expression***)**

Parameters

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

Return value

String

Comments

The following table lists the expression types in Dexterity and the corresponding string representation:

Expression type	String representation
Numeric	The text representation of the value. No formatting is included.
Boolean	False is converted to the character 0. True is converted to the character 1.
Date and Time	The text representation of the date or time value, with formatting specified by the current system settings.

If you are converting a currency value to a string, you may want to use the ***format()*** *function, which allows you to specify the formatting for the currency value.*

Examples

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

```
warning "Amount " + str(Total) + " is above the limit.";
```

Related items

Commands

format(), **value()**

substitute

Description

The **substitute** statement substitutes specified values for replacement markers in the specified string.

Syntax

substitute *string, string_expr* {, *addl_string_expr*}

Parameters

- *string* – Any string that can be set, containing the string that substitute strings will be added to. This string contains replacement markers (%1, %2, %3 and so on) that will be replaced by the values of the *string_expr* and *addl_string_expr* parameters. For instance, the replacement marker in the string "Account %1 doesn't exist." would be replaced by a *string_expr* value, resulting in a string such as "Account 001-4432-05 doesn't exist."
- *string_expr* – The first string expression to substitute for the %1 replacement marker.
- *addl_string_expr* – Additional string expressions to substitute for the additional replacement markers, %2, %3 and so on, in *string*.

Comments

If there are more parameters than replacement markers, the extra parameter strings are ignored. Replacement markers that don't have items substituted for them will appear in the string.

The **substitute** statement is useful for customizing message strings by specifying the exact problem that has occurred.

Examples

The following example inserts a value into a string to alert a user that a customer record doesn't exists. The Customer Name field has the value "Greg Smith".

```
local string message;

message = "The customer %1 is not found. Please %2.";
substitute message, 'Customer Name', "choose another";
warning message;
{Dialog box displays "The customer Greg Smith is not found.
Please choose another."}
```

Related items

Commands

getmsg()

Additional information

Chapter 18, "Messages," in Volume 1 of the Dexterity Programmer's Guide

substring()

Description The **substring()** function returns a portion of a specified string or text field.

Syntax **substring(***source*, *begin*, *length***)**

Parameters

- *source* – A string or text variable, field or value containing the substring you want.
- *begin* – An integer indicating the starting position within the *source*.
- *length* – An integer indicating the number of characters you want to return from the source string. You can return a maximum of 255 characters at a time.

Return value String

Examples The following examples return a substring from the string "This is a test". In this example, the function begins at the first character and returns the next four characters, so new_string is set to "This".

```
new_string = substring("This is a test", 1, 4);
```

The following example begins at the eleventh position and returns the next six characters. Since there are only four characters after the eleventh position, only those four characters are returned, and new_string will be set to "test".

```
new_string = substring("This is a test", 11, 6);
```

The following example specifies that the function start at position 20, which is beyond the end of the source string. Nothing is returned and the new_string variable will be empty.

```
new_string = substring("This is a test", 20, 5);
```

The following example returns the first 20 characters from the text field Comment to the local variable l_comment.

```
local string l_comment;

l_comment = substring('Comment', 1, 20);
```

Related items

Commands

pos()

sum range

Description

The **sum range** statement totals each field that has a numeric data type (integer, long integer and currency) in a specified range.

Syntax

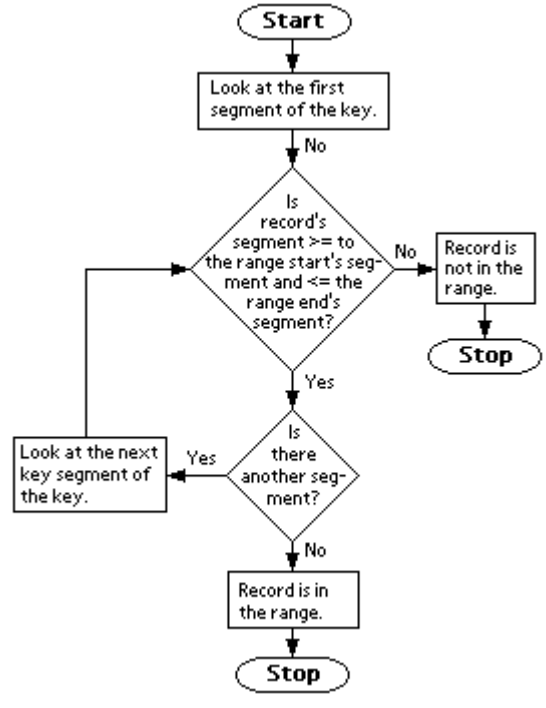
sum range table *table_name*

Parameters

- **table** *table_name* – The name of the table containing the fields to be summed.

Comments

You must use the **range** statement to specify the range that will be used by the **sum range** statement. The **sum range** statement creates an *exclusive* range from the values specified by the **range** statement. In an exclusive range, each record's key segments are evaluated individually, comparing each segment's value to the value specified when the range was created. If any key field for a record is outside of the specified range for that key segment, the record will be excluded from the range used by the **sum range** statement. The following flowchart describes the process.



The **sum range** statement only totals numeric fields (integer, long integer and currency) that have non-zero values in the table buffer. After you establish the range using the **range** statement, you should specify which fields you want to sum by clearing the table buffer and setting the value of each of these fields to 1. You can then issue the **sum range** statement to total these fields.

*Using the **sum range** statement instead of a loop construct can improve the speed and efficiency of applications that use the SQL database type.*

For example, you could have a key for the Invoice table that includes two key segments, Invoice_Number and Item_Number. If you use that key to create a range of all invoice numbers between 100 and 110, and all item numbers between 1 and 5, the following records would be included in the range created by the **range** statement for the Pervasive.SQL or c-tree database managers.

Notice the record for invoice number 102. It's included in the range, even though its corresponding item number is 6. It's included because for the Pervasive.SQL and c-tree database managers, the **range** statement creates an *inclusive* range. Refer to the **range** statement for a complete description of how ranges are evaluated for the c-tree, Pervasive.SQL and SQL database managers.

If you use the **sum range** statement to sum the Item_Price field, the statement would first create an exclusive range based upon the range shown in the illustration. The record for Invoice 102 would not be included in this exclusive range, because even though 102 is within the specified range for the first key segment, 6 is outside of the defined range for the second key segment.

In the previous example, the summed value for the Item_Price field returned by the **sum range** statement would be $141.50. If you were using the Pervasive.SQL or c-tree database managers and had set a range only and used a loop construct to loop through the records in the range totaling the Item_Price field, the result would have been $183.50.

*Keep these range differences in mind if you plan to replace loop constructs in existing applications with the sum range statement. Using the **clear field** and **fill** statements for all key segments after the first key segment is one way of ensuring that the ranges used by the **range** and **sum range** statements are the same for all database types.*

Overflow situations

Depending upon the database type used, Dexterity responds differently when an overflow situation is encountered. If a SQL database type is used, an error is returned and no values will be summed. If the c-tree Plus or Pervasive.SQL database types are used, no error will be returned. Instead, currency fields will be returned at their maximum value, while integer and long integer fields will wrap within their allowable range of values. For example, if the sum of integer fields should be 33,750, the value -31,786 will be returned instead.

Be sure to test for overflow situations, especially if you are summing integer fields, large ranges, or fields with large numeric values.

Examples

The following example sets a range on a table using a key containing only one segment. It sets a range on the customer_info table based upon the customer ID. It then clears the table buffer and sets the fields to be summed, the currency field maint_fee and the integer field number_of_machines, to 1 before using the **sum range** statement.

```
{Clear the current range and set the range to include customers served
by repair team 1: customer IDs 1 to 100.}
range clear table customer_info;
cust_ID of table customer_info = 1;
range start table customer_info by ID_key;
cust_ID of table customer_info = 100;
range end table customer_info by ID_key;
{Clear the current table buffer.}
clear table customer_info;
```

```
{Initialize the values of the fields to be summed to 1.}
maint_fee of table customer_info = 1;
number_of_machines of table customer_info = 1;
sum range table customer_info;

{Display the sum information in a warning dialog box.}
warning "Repair team 1 generates maintenance fees of " +
➥ str(maint_fee of table customer_info) +
➥ "and they are responsible for maintaining " +
➥ str(number_of_machines of table customer_info) + "machines.";
```

The following example sets a range on a table using a key containing two segments. It sets a range on the Invoice table based upon the first key, which contains the Invoice_Number and Item_Number key segments. Since the range should include all item numbers for invoice number 10002, this script uses the **clear field** and **fill** statements when setting the range for the second key element's values. In this case, the range used for the **sum range** statement will be the same range as the one established by the **range** statement.

```
range clear table Invoice;
{Set the beginning value for both key segments.}
Invoice_Number of table Invoice = 10002;
clear field Item_Number of table Invoice;
range start table Invoice;
{Set the ending value for the Item_Number to its maximum value. Since
the value for the Invoice_Number field doesn't change, don't set an
ending value for that key segment.}
fill Item_Number of table Invoice;
range end table Invoice;
clear table Invoice;

{Initialize the value of the field to be summed to 1.}
Item_Cost of table Invoice = 1;
sum range table Invoice;
warning "The subtotal for Invoice 10002 is" + str(Item_Cost of table
➥ Invoice);
```

Related items

Commands
clear field, **fill**, **range**

sysdate()

Description

The **sysdate()** function returns the current date, from either the current computer or the server on which the data source the current computer is accessing resides.

Syntax

sysdate({CURRENT_SERVER})

Parameters

- **CURRENT_SERVER –** This keyword applies only when a SQL database type is used. It is a constant representing the server the current computer is connected to, regardless of what that server's name is.

Return value

A date or datetime value

Comments

This function will return the system date of the current computer if the **CURRENT_SERVER** keyword isn't used, or if the current computer isn't connected to a data source.

Examples

The following example sets the value of the Start_Date field to the system date of the server on which the current SQL data source resides.

```
Start_Date = sysdate(CURRENT_SERVER);
```

The following example sets the value of the Start_Date field to the system date of the current computer.

```
Start_Date = sysdate();
```

Related items

Commands

addmonth(), day(), dow(), eom(), mkdate(), month(), setdate(), year()

sysdatetime()

Description

The **sysdatetime()** function returns the current date and time from the current computer.

Syntax

sysdatetime()

Parameters

- None

Return value

A datetime value

Examples

The following example sets the value of the Start_DateTime field to the system date and time of the current computer.

```
Start_DateTime = sysdatetime();
```

Related items

Commands

addmonth(), **day()**, **dow()**, **eom()**, **mkdate()**, **month()**, **setdate()**, **sysdate()**, **systime()**, **year()**

system

Description

The **system** statement performs system commands.

Syntax

system *expr*, *parameter*, *parameter*...

Parameters

- *expr* – An integer or its constant specifying the system command to be executed.
- *parameter* – A value passed to the system command.

Comments

Prior to Release 3.0 of Dexterity, the tasks performed by the functions described in the Function Libraries were performed by system commands. While we now recommend that you use Function Library functions instead of system commands in your applications, this statement continues to be supported for backward compatibility. Thus, applications created using earlier versions of Dexterity will still compile properly using the latest version of Dexterity.

Examples

The following example shows the system command that sets the About menu item.

```
system 2520, "Time and Billing";
```

Related items

Additional information

Function Library Reference manual

systime()

Description

The **systime()** function returns the current system time from either the current computer or the server on which the data source the current computer is accessing resides.

Syntax

systime({CURRENT_SERVER})

Parameters

- **CURRENT_SERVER –** This keyword applies only when a SQL database type is used. It is a constant representing the server the current computer is connected to, regardless of what that server's name is.

Return value

A time or datetime value

Comments

This function will return the system time of the current computer if the **CURRENT_SERVER** keyword isn't used, or if the current computer isn't connected to a data source.

Examples

The following example sets the value of the Print Time field to the system time of the server on which the current SQL data source resides.

```
'Print Time' = systime(CURRENT_SERVER);
```

The following example sets the value of the Print Time field to the system time of the current computer.

```
'Print Time' = systime();
```

Related items

Commands

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

technicalname()

Description

The **technicalname()** function returns the technical name for the specified resource.

Syntax

technicalname(*resource* {, *qualifier*}**)**

Parameters

- *resource* – The resource whose technical name is being returned.
- *qualifier* – An optional integer constant that specifies whether qualifiying text is returned with the name. Use one of the following constants:

Constant	Description
QUALIFY_NONE	The technical name is returned without any qualifiers.
QUALIFY_PREFIX	The technical name is returned along with any qualifiers that would appear before the name.
QUALIFY_SUFFIX	The technical name is returned along with any qualifiers that would appear after the name.
QUALIFY_FULL	The technical name is returned with any qualifiers that would appear before or after the name.

If this parameter isn't supplied, the technical name is returned without any qualifiers.

Comments

Some sanScript statements and functions require you to supply technical names of resources. You can use the **technicalname()** function to retrieve technical names, rather than hard-coding technical names in your scripts.

The following table lists the prefix and suffix values that can be returned for the various types of resources:

Resource type	Prefix	Suffix
Form	form	N/A
Window	window	of form <formname>
Scrolling window	window	of form <formname>
Window field	field	of window <windowname> of form <formname>
Menu	menu	of form <formname>
Report	report	N/A
Table	table	N/A

Resource type	Prefix	Suffix
Table field	field	of table <tablename>
Table group	tablegroup	N/A
Global procedure	script	N/A
Global function	function	N/A
Form-level procedure	script	of form <formname>
Form-level function	function	of form <formname>
Global variable	field	of globals

Examples

The following example retrieves the fully-qualified technical name for the Monthly Utility Cost window field.

```
local string resource_name;

resource_name = technicalname('Monthly Utility Cost', QUALIFY_FULL);
```

The following example retrieves the technical name of the House_Data table.

```
local string resource_name;

resource_name = technicalname(table House_Data, QUALIFY_PREFIX);
```

Related items

Commands
physicalname()

this

Description

The **this** statement is used to reference the current callback object from within a method defined for that callback object.

Syntax

this.{*property* | *method*}

Parameters

- *property* | *method* – A property or method to access for the current callback object.

Examples

The following example is the VerifyPassword() user-defined function that runs in response to a method in a callback object. It uses the **this** statement to set the value of the **Validated** parameter for the current callback object.

```
function returns boolean OK;

in string password;

if password = Password of globals then
    OK = true;
    this.Validated = true;
else
    OK = false;
    this.Validated = false;
end if;
```

Related items

Additional information

Chapter 40, "Callbacks and Events," in Volume 2 of the Dexterity Programmer's Guide

throw

Description

The **throw** statement is used to throw a user exception or to rethrow the current exception.

Syntax

throw {*class*, *subclass*, *message*}

Parameters

- *class* – A long integer specifying the class of the user exception. This value should be 1 or greater. For integrating applications it should be 22,000 or greater.
- *subclass* – A long integer specifying the subclass of the user exception.
- *message* – A string specifying the message that describes the user exception. You can use the following string constants in the message to indicate where the exception occurred.

Constant	Description
script	The name of the script in which the exception was generated.
line	The line number in the script where the exception was generated.

Comments

If your exception handler has received an exception not intended for it, use the **throw** statement without any parameters to rethrow the exception.

Examples

The following example is part of a procedure that verifies records in the Buyer_Data table. If the current record in the Buyer_Data table doesn't have a buyer name, the **throw** statement is used to throw a user exception.

```
in table Buyer_Data;

if 'Buyer Name' of table Buyer_Data = "" then
    throw BUYER_DATA, NO_NAME, "Buyer ID " + 'Buyer ID' of table
    ➥ Buyer_Data + " has an invalid name.";
end if;
```

Related items

Commands

restart script, **restart try**, **try...end try**

Additional information

Chapter 26, "Structured Exception Handler," in Volume 2 of the Dexterity Programmer's Guide

throw system exception for table

Description

The **throw system exception for table** statement is used to throw a system exception for an error that has occurred for a table.

Syntax

throw system exception for table *table_name*

Parameters

- *table_name* – Specifies the name of the table for which the error status of the last operation will be checked.

Comments

In versions of Dexterity prior to release 5.5, the **check error** statement was used to throw exceptions for table errors. The **check error** statement no longer throws exceptions.

Examples

The following example attempts to read and actively lock a record from the Seller_Data table. Note that the **throw system exception for table** statement is used to throw a system exception if a database error occurred.

```
local integer read_count = 0;

try
    'Seller ID' of table Seller_Data = 'Seller ID' of window Sellers;
    {Attempt to actively lock the record.}
    change table Sellers, lock;
    increment read_count;
    {If an error occurred, an exception will be thrown.}
    throw system exception for table Seller_Data;

catch [EXCEPTION_CLASS_DB]
    {A standard database error occurred.}
    if err() = LOCKED then
        {Retry the read operation 10 times.}
        if read_count < 10 then
            restart try;
        else
            error "Unable to read and actively lock the record.";
        end if;
    end if;
else
    {Some other exception occurred.}
    error "An unknown exception occurred. Class: " +
    ➥ str(Exception_Class()) + " Subclass: " +
    ➥ str(Exception_SubClass());
end try;
```

transaction begin

Description

The **transaction begin** statement starts a database transaction. All table operations that occur after the **transaction begin** statement are part of the transaction. The transaction is completed using the **transaction commit** statement or discarded using the **transaction rollback** statement.

Syntax

transaction begin

Parameters

None

Comments

Transaction processing capability isn't available with all database types. Its availability can be checked using the **havetransactions()** function.

Examples

The following example uses the **transaction begin** statement to begin a series of table operations for a transaction. In this case, the status for all of the line items in an invoice is being updated. If all of the items are successfully updated, the transaction is committed using the **transaction commit** statement. If an unexpected error occurs during the transaction, the **transaction rollback** statement is used to roll the transaction back.

```
local boolean transaction_started;
local integer retry_count;
local integer MAXRETRIES = 10;
local long sleeptime;

{Set up the range used for the transaction.}
'Invoice Number' of table Invoice_Lines = 'Invoice Number' of window
➥ Invoices;
range start table Invoice_Lines;
range end table Invoice_Lines;

retry_count = 0;
try
    {If a transaction can be used, start one.}
    if (Tools_GetTranLevel() = 0) and (havetransactions()) then
        transaction begin;
        transaction_started = true;
    else
        transaction_started = false;
    end if;
```

```
    release table Invoice_Lines;
    change first table Invoice_Lines;
    while err() = OKAY do
        'Post Flag' of table Invoice_Lines = true;
        save table Invoice_Lines;
        change next table Invoice_Lines;
    end while;

    {If a transaction has been started by this code, end it.}
    if transaction_started = true then
        transaction commit;
    end if;

catch [EXCEPTION_CLASS_DB_DEADLOCK]
    {If a transaction was started, it was automatically rolled back.}

    if transaction_started = true then
        {If this code started the transaction, retry it until the
         retry limit is exceeded.}
        if retry_count < MAXRETRIES then
            sleeptime = Timer_Sleep(10);
            increment retry_count;
            restart try;
        else
            {The retry count was exceeded. Rethrow the exception.}
            throw;
        end if;
    else
        {A deadlock exception occurred, but it wasn't for the
         transaction that this code started. Rethrow the exception.}
        throw;
    end if;
else
    {Some other unexpected exception occurred. If the transaction
     this code started is still pending, roll it back.}
    if transaction_started = true then
        transaction rollback;
    end if;
    {Display the exception.}
    error "An unknown exception occurred. Class: " +
    ➥ str(Exception_Class()) + " Subclass: " +
    ➥ str(Exception_SubClass());
end try;
```

Related items

Commands

havetransactions(), **transaction commit**, **transaction rollback**

Additional information

Chapter 19, "Transactions," in Volume 2 of the Dexterity Programmer's Guide

transaction commit

Description

The **transaction commit** statement ends a transaction. The table changes made within the transaction are "committed," making them permanent.

Syntax

transaction commit

Parameters

None

Comments

Table operations can be undone by using the **transaction rollback** statement instead of a **transaction commit**. Transaction processing capability isn't available with all database types and its availability can be checked using the **havetransactions()** function.

Examples

Refer to the example for **transaction begin**.

Related items

Commands

havetransactions(), **transaction begin**, **transaction rollback**

Additional information

Chapter 19, "Transactions," in Volume 2 of the Dexterity Programmer's Guide

transaction rollback

Description

The **transaction rollback** statement is used to "roll back" the table operations that have occurred since the **transaction begin** statement.

Syntax

transaction rollback

Parameters

None

Comments

Transaction rollback isn't available with all database types, and its availability can be checked using the **havetransactions()** function.

Examples

Refer to the example for **transaction begin**.

Related items

Commands

havetransactions(), **transaction begin**, **transaction commit**

Additional information

Chapter 19, "Transactions," in Volume 2 of the Dexterity Programmer's Guide

trim()

Description

The **trim()** function removes trailing blanks from a specified string. It can also be used to remove a specified string from the beginning, end or both the beginning and end of another specified string.

Syntax

trim(*string* {, *direction*, *trimstring*}**)**

Parameters

- *string* – A string field or string value from which to remove blanks or a specified string.
- *direction* – A constant that specifies where the *trimstring* should be removed from the *string*. The possible values for the parameter are:

Constant	Description
LEADING	Remove *trimstring* from the start of *string*.
TRAILING	Remove *trimstring* from the end of *string*.
LEADING+TRAILING	Remove *trimstring* from the beginning and end of *string*.

- *trimstring* – A string field or value to be removed from *string*.

If the trimstring *and* direction *parameters aren't specified, spaces will be trimmed from both the beginning and the end of the* string.

Return value

String

Examples

The following example removes the excess spaces from the beginning and end of the Customer_Name field, and returns the trimmed string value to the same field.

```
Customer_Name = trim(Customer_Name);
```

The following example removes the backslash from both the beginning and end of the string "/drive:directory/" and places the resulting string in a local string field.

```
string1 = trim("/drive:directory/", LEADING+TRAILING, "/");
```

Related items

Commands

pad(), **substring()**

truncate()

Description

The **truncate()** function returns the truncated value of a specified currency or variable currency field. The field can be truncated on either side of the decimal separator. The truncated digits are replaced with zeros.

Syntax

truncate(*field_name*, *side*, *expression***)**

Parameters

- *field_name* – A currency field or value containing the value to truncate.
- *side* – A constant indicating on which side of the decimal separator you want to begin truncating:

Constant	Description
DECIMALRIGHT	Truncate the currency value from the specified number of places to the right of the decimal separator.
DECIMALLEFT	Truncate the currency value from the specified number of places left of the decimal separator.

- *expression* – An integer indicating the number of places to the right or left of the decimal separator that the specified field will be truncated to. For currency fields, the allowable range to the right of the decimal separator is 0 to 5. For variable currency fields, it is 0 to 15.

Return value

Currency or variable currency

Examples

The following example truncates 1050.123 to 1050.120, or two places to the right of the decimal separator.

```
a = truncate(1050.123, DECIMALRIGHT, 2);
```

The following example truncates the Amount field, containing the value 1050.12, to two places to the left of the decimal separator or 1000.00.

```
a = truncate(Amount, DECIMALLEFT, 2);
```

The following example truncates 1050.1238 to one decimal digit more than the number specified in the user's operating system's decimal digits setting. In this example, the **currencydecimals()** function is used to return the current value of this operating system setting (which is 2). Therefore, 1050.1238 will be truncated three decimal places to the right of the decimal separator. The value returned will be 1050.123.

```
a = truncate(1050.1238, DECIMALRIGHT, currencydecimals() + 1);
```

Related items

Commands

currencydecimals(), **round()**, **set precision of**

try...end try

Description

The **try...end try** statement is used to implement structured exception handling in Dexterity.

Syntax

try *statements* {**catch** [*class*] *statements*} {**else** *statements* } **end try**

Parameters

- *statements* – The sanScript statements to be executed.
- **catch** [*class*] – A long integer specifying the class of the exception to catch. A **try...end try** statement can contain several **catch** clauses. If a **catch** clause will catch several classes of exceptions, separate them with commas.
- **else** – If the **else** clause is included, then the statements following it will be run if none of the **catch** clauses catch the exception that was thrown.

Comments

The **try...end try** statement must contain at least one **catch** clause or an **else** clause.

If none of the **catch** clauses catch the exception and there is no **else** clause, the exception is considered *unhandled.* A dialog box is automatically displayed describing the situation to the user.

Examples

The following example uses the **try...end try** statement to handle the exception that could occur if the result of the **substitute** statement is longer than 255 characters. The **catch** clause catches the overflow exception. Notice that the _script_ constant is used to include the name of the script in the message. Any other exception will be handled by the **else** clause.

```
local string error_message;

try
    error_message = getmsg(22001);
    substitute error_message,"could not be launched","choose another";
    error error_message;
catch [EXCEPTION_CLASS_SCRIPT_STRING_OVERFLOW]
    {Catch the exception for a string overflow.}
    error "String overflow on substitute in script" + _script_ + ".";
else
    {Some other exception occurred.}
    error "An unknown exception occurred. Class: " +
    ➥ str(Exception_Class()) + " Subclass: " +
    ➥ str(Exception_SubClass());
end try;
```

Related items

Commands

exit, **restart script**, **restart try**, **throw**

Additional information

Chapter 26, "Structured Exception Handler," in Volume 2 of the Dexterity Programmer's Guide

uncheck command

Description

The **uncheck command** statement sets a command to the unchecked state. The command will appear unmarked in any menu or toolbar in which it is displayed.

Syntax

uncheck command *command_name* {, *command_name*}

Parameters

- *command_name* – The name of the command to be unchecked.

Comments

Use the **Command_GetBooleanProperty()** function to find out whether a command is checked or unchecked.

Examples

The following example unchecks the CMD_ComputeTaxes command. The command's appearance will be changed any place the command is displayed, such as a menu or toolbar.

```
uncheck command CMD_ComputeTaxes;
```

Related items

Commands

check command

Additional information

Chapter 15, "Commands," in Volume 1 of the Dexterity Programmer's Guide

uncheck field

Description

The **uncheck field** statement removes the highlighting from a specified item in a multi-select list box or unchecks an item in a button drop list.

Syntax

uncheck field *field_name*, *number*

Parameters

- *field_name* – The name of the field containing the item you want deselected or unmarked.
- *number* – An integer indicating which item to deselect or unmark.

Comments

The **finditem()** function can be used to retrieve the number of an item from a multi-select list box or any other list field. The retrieved number can then be used as the *number* parameter.

Examples

The following example removes the highlight from the first item in the Payment Methods multi-select list box.

```
uncheck field 'Payment Methods', 1;
```

The following example is the change script for a button drop list. It uses the **checkedfield()** function to find out whether the selected item was checked. If it was, it removes the check mark next to the item that was selected. Otherwise, it checks the item.

```
if checkedfield('Reports', 'Reports') then
    uncheck field 'Reports', 'Reports';
else
    check field 'Reports', 'Reports';
end if;
```

Related items

Commands
check field, **finditem()**

uncheck menu

Description

The **uncheck menu** statement removes a check mark next to a menu item.

Syntax

uncheck menu *menu_name*, *number*

Parameters

- *menu_name* – The name of the menu containing the item you want unchecked.
- *number* – An integer representing the placement of the item to be unchecked in the menu.

Comments

The **uncheck menu** statement cannot be used to remove a check mark next to grouped menu items that the application inherited, such as the Clipboard menu items Cut, Copy and Paste. For more information about menu items that can be checked and unchecked, refer to Chapter 14, "Form-based Menus," in Volume 1 of the Dexterity Programmer's Guide.

Examples

The following example removes a check mark next to the third item in the Transactions menu.

```
uncheck menu Transactions, 3;
```

Related items

Commands

check menu, **checkedmenu()**

unlock

Description

The **unlock** statement unlocks one or more window fields that were previously locked with the **lock** statement. Unlocking a field allows the user to enter data in that field.

Syntax

unlock {**field**} *field_name* {, *field_name, field_name*...}

Parameters

- **field** – An optional keyword that identifies *field_name* as a field.
- *field_name* – The name of a field to be unlocked.

Comments

Fields with the Editable property set to false in the Properties window *can't* be unlocked using the **unlock** statement.

Examples

The following example unlocks the Customer ID field.

```
unlock field 'Customer ID';
```

The following example unlocks the Customer ID, Company Name and Contact Name fields.

```
unlock 'Customer ID', 'Company Name', 'Contact Name';
```

Related items

Commands

lock

upper()

Description

The **upper()** function returns a string in which all the alphabetic characters of a specified string have been converted to uppercase.

Syntax

upper(*string_expression***)**

Parameters

- *string_expression* – The string or string field you wish to convert to upper case.

Return value

String

Comments

This function can be used to manipulate a string without regard to the case of the characters in the string.

Examples

The following example evaluates the contents of the Password field without regard to the case of the characters in the string.

```
if upper(Password) = 'Password' of table User_Passwords then
    open window Customer_Maintenance;
end if;
```

Related items

Commands

lower()

value()

Description

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

Syntax

value(*string***)**

Parameters

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

Return value

Initially, a variable currency value is returned. If necessary, the value is converted to the appropriate storage type based upon the storage type of the field or variable the value is set to. If a string contains no numeric sequence, a value of 0 will be returned.

Comments

Only the first numeric sequence in the string will be converted. For example, the string "Jones78Smith8" will be converted to 78.

If the string value contains a numeric sequence with the system-defined decimal separator, the return value may be rounded, depending on the storage type the value is returned to. If returned to an integer or long integer, the return value will be rounded up or down as appropriate. The return value won't be rounded if it is returned to a currency or varaible currency storage type.

Examples

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

```
new_integer = 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 field.

```
new_integer = value("Cost123.45");
```

The following example converts the string "&hFF" (representing the hexadecimal value FF) to its numeric equivalent.

```
int_value = value("&hFF");
```

Related items

Commands

hex(), **str()**

Additional information

Hexadecimal values on page 55 of Volume 2 the Dexterity Programmer's Guide

warning

Description

The **warning** statement creates a warning dialog box displaying the specified string.

Syntax

warning *expression* {**with help number** *context_number*}

Parameters

- *expression* – A string field, text field or string or text value with the message to be displayed in the dialog box.
- **with help number** *context_number* – A long integer specifying a help context number associated with a specific topic in the online help file for the current dictionary. If this parameter is used, a button labeled Help will appear in the dialog box. If a user presses the Help button, the specified help file topic will be displayed. Refer to Chapter 7, "Windows Help," in the Dexterity Stand-alone Application Guide for more information.

Comments

The warning dialog box will have one button labeled OK. The window closes automatically after the user clicks OK. The icon displayed is the standard warning icon for the given operating system. The warning dialog box is shown in the following illustration.

Examples

The following example displays a message in a warning dialog box, including the value for the Customer ID field. Note that a help context number (represented by the constant billion + 81) has been added. If the user clicks Help, the topic associated with that number will be displayed.

```
warning "The customer_" + 'Customer ID' + " doesn't have a valid
➥ address." with help number billion + 81;
```

Related items

Commands

ask(), **error**, **getmsg()**, **getstring()**, **substitute**

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 the 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.

Examples

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

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

Related items

Commands

continue, **exit**, **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 or datetime value.

Return value

Integer

Examples

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;

year = year(sysdate());
```

Related items

Commands

addmonth(), **day()**, **dow()**, **eom()**, **mkdate()**, **month()**, **setdate()**, **sysdate()**

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y