Function Reference

VERSION 4.0

PowerBuilder

December 1994

Contents

About This Manual

Subject

The bulk of this manual describes the PowerScript functions that are provided with PowerBuilder. The syntax of each function is explained and examples are provided. In addition, the manual describes the functions you can use in DataWindow painter expressions and how to construct text-matching expressions. Two appendixes describe the attributes and syntax for DataWindows.

For lists of the functions that are part of the definition of a PowerBuilder object, see *Objects and Controls*. For categorized lists of functions with syntax, see the *Quick Reference*.

Audience

This manual is for programmers who will be using PowerBuilder to build client/server applications.

Online Help

When you have a question about using PowerBuilder, you can access its extensive online Help system. By accessing Help you can see:

- **Procedures** for accomplishing tasks in PowerBuilder
- **Reference information** about PowerBuilder topics or components
- **Context-sensitive information** about PowerBuilder functions or reserved words in scripts

For more information on getting Help in PowerBuilder, see the *User's Guide*.

CHAPTER 1

PowerScript Functions

This chapter provides syntax, descriptions, and examples for the PowerScript functions. The functions are listed in alphabetical order.

Platform information
You can use the functions in this chapter in any event script. Because of differences in platforms, some functions have no effect on a particular platform. This is noted in the function or argument descriptions. To write scripts that are compatible across platforms, you can include these functions in your scripts. However, you should include code to check their return values and take appropriate action on each platform.

Abs

Description

Calculates the absolute value of a number.

Syntax

Abs (*n*)

Parameter	Description
n	The number for which you want the absolute value

Return value

The data type of *n*. Returns the absolute value of *n*.

Examples

All these statements set num to 4:

```
integer i, num
i = 4
num = Abs(i)
num = Abs(4)
num = Abs(+4)
num = Abs(-4)
```

This statement returns 4.2:

```
Abs(-4.2)
```

See also

Abs in Chapter 2, "DataWindow Painter Functions"

AcceptText

Description

Applies the contents of the DataWindow's edit control to the current item in the buffer of a DataWindow control. The data in the edit control must pass the validation rule for the column before it can be stored in the item.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**AcceptText** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to accept data entered in the edit control

Return value

Integer. Returns *1* if it succeeds and *-1* if it fails (for example, the data did not pass validation).

Usage

When a user moves from item to item in a DataWindow control, the control validates and accepts data the user has edited. When a user modifies a DataWindow item then immediately changes focus to another control in the window, the DataWindow control does not accept the modified data—the data remains in the edit control. Use the AcceptText function in this situation to ensure that the DataWindow object contains the data the user edited. A typical place to call AcceptText is in a user event that is posted from the DataWindow's LoseFocus event.

> **AcceptText and stack faults**
> Never call AcceptText in the ItemChanged event. Because AcceptText can trigger the ItemChanged event, such a recursive call can cause a stack fault.

Events

AcceptText may trigger an ItemChanged or an ItemError event.

Examples

In this example, the user is expected to enter a key value (such as an employee number) in a column of the DataWindow object, then click the OK button. This script for the Clicked event for the button calls AcceptText to validate the entry and place it in the DataWindow control. Then the script uses the item in the Retrieve function to retrieve the row for that key:

```
IF dw_emp.AcceptText( ) = 1 THEN
   dw_emp.Retrieve(dw_emp.GetItemString  &
      (dw_emp.GetRow( ),dw_emp.GetColumn( )))
END IF
```

This script for the Clicked event for a CommandButton accepts the text in the DataWindow dw_Emp and counts the rows in which the column named balance is greater than 0:

```
Integer i, Count
dw_employee.AcceptText( )
FOR i = 1 to dw_employee.RowCount( )
   IF dw_employee.GetItemNumber(i,'balance')   &
         > 0 THEN
      Count = Count + 1
   END IF
NEXT
```

This script for the Clicked event for a CommandButton accepts the text in the child DataWindow.

```
DataWindowChild dwc
integer rtncode

rtncode = dw_emp.GetChild("emp_id", dwc)

// Statements to check for errors
dwc.SetTransObject(SQLCA)
dwc.Retrieve("argument")
... //Some processing
dwc.AcceptText( )
```

See also Update

Activate

Description Activates the object in an OLE 2.0 control, allowing the user to work with the object within the control or in the server application.

Syntax *ole2control.***Activate** (*activationtype*)

Parameter	Description
ole2control	The name of the OLE 2.0 control that contains the object you want to activate

Parameter	Description
activationtype	A value of the enumerated data type omActivateType specifying where the user will work with the OLE object. Values are: ♦ InPlace! — The object is activated within the control. The subset of menus provided by the server application are merged with the PowerBuilder application's menus. ♦ OffSite! — The object is activated in the server application, which gives the user access to more of the server application's functionality

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Control is empty
- *-2* Invalid verb for object
- *-3* Verb not implemented by object
- *-4* No verbs supported by object
- *-5* Object can't execute verb now
- *-9* Other error

Example

This example activates the object in ole_1 in the server application:

```
integer result
result = ole_1.Activate(OffSite!)
```

See also

DoVerb
OLEActivate
SelectObject

AddCategory

Description

Adds a new category to the category axis of a graph. AddCategory is for a category axis whose data type is string.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**AddCategory** (*categoryname*)

Parameter	Description
controlname	The name of the graph to which you want to add a category.
categoryname	A string whose value is the name of the category you want to add to *controlname*. The category will appear as a label on the category axis.

Return value

Integer. Returns the number assigned to the category if it succeeds and if *categoryname* already exists as a label on the category axis, AddCategory returns the number of the existing category. Returns *-1* if an error occurs.

Usage

AddCategory adds a category to the end of the category axis. The category becomes an "empty slot" in each series to which you can assign a data point. A tick mark exists on the category axis for all the categories associated with the graph.

When the data type of the category axis is string, you can specify the empty string ("") as the category name. However, because category names must be unique, there can be only one category with that name. Also, category names are unique if they have different capitalization.

To add categories when the axis data type is date, DateTime, number, or time, use InsertCategory. Also, to insert a category in the middle of a series, use InsertCategory. You can also use InsertCategory to add a category to the end of a series, as AddCategory does, although it requires an additional argument to do it.

To add data to a series in the graph, use the AddData or InsertData function. You can add a data value and put it in a new category, or you can add or change data in an existing category. To add a series to the graph, use the AddSeries function.

Examples

This statement adds a category named PCs to the graph gr_product_data:

```
gr_product_data.AddCategory("PCs")
```

See also

AddData
AddSeries

DeleteData
DeleteSeries

AddData

Description

Adds a value to the end of a series of a graph. The syntax you use depends on the type of graph:

- For all graph types except scatter graphs, use Syntax 1 to specify a data value and a category.
- For scatter graphs only, use Syntax 2 to specify an x and y value for the data point.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

controlname.**AddData** (*seriesnumber*, *datavalue* {, *categoryvalue* })

Parameter	Description
controlname	The name of the graph in which you want to add data to a series. The graph's type should not be scatter.
seriesnumber	The number that identifies the series to which you want to add data
datavalue	The value of the data you want to add
categoryvalue (optional)	The category for this data value on the category axis. The data type of the *categoryvalue* should match the data type of the category axis. In most cases, you should include *categoryvalue*. Otherwise, an uncategorized value will be added to the series.

Return value 1

Long. Returns the position of the data value in the series if it succeeds and *-1* if an error occurs.

Syntax 2

controlname.**AddData** (*seriesnumber*, *xvalue*, *yvalue*)

Parameter	Description
controlname	The name of the scatter graph in which you want to add data to a series. The graph's type should be scatter.
seriesnumber	The number that identifies the series to which you want to add data
xvalue	The x value of the data point you want to add
yvalue	The y value of the data point you want to add

Return value 2

Long. Returns the position of the data value in the series if it succeeds and *-1* if an error occurs.

Usage

When you use Syntax 1, AddData adds a value to the end of the specified series or to the specified category, if it already exists. If *categoryvalue* is a new category, the category is added to the end of the series with a label for the data point's tick mark. If the axis is sorted, the new category is incorporated into the existing order. If the category already exists, the new data replaces the old data at the data point for the category.

For example, if the third category label specified in series 1 is March and you add data in series 4 and specify the category label March, the data is added at data point 3 in series 4.

When the axis data type is string, you can specify the empty string ("") as the category name. However, because category names must be unique, there can be only one category with a blank name. If you use AddData to add data without specifying a category, you will have data points without categories, which is not the same as a category whose name is "".

To insert data in the middle of a series, use InsertData. You can also use InsertData to add data to the end of a series, as AddData does, although it requires an additional argument to do it.

For a comparison of AddData, InsertData, and ModifyData, see Equivalent syntax in InsertData.

Examples

These statements add a data value of 1250 to the series named Costs and assign the data point the category label Jan in the graph gr_product_data:

```
integer SeriesNbr
// Get the number of the series.
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.AddData(SeriesNbr, 1250, "Jan")
```

These statements add a data value of 1250 to the end of the series named Costs in the graph gr_product_data but do not assign the data point to a category:

```
integer SeriesNbr
// Get the number of the series.
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.AddData(SeriesNbr, 1250)
```

Syntax 2

These statements add the x and y values of a data point to the series named Costs in the scatter graph gr_sales_yr:

```
integer SeriesNbr
// Get the number of the series.
SeriesNbr = gr_sales_yr.FindSeries("Costs")
gr_sales_yr.AddData(SeriesNbr, 12, 3)
```

See also

DeleteData
FindSeries
GetData
InsertData

AddItem

Description

Adds a new item to the list of values in a listbox.

Applies to

ListBox, DropDownListBox

Syntax

listboxname.**AddItem** (*item*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want to add an item
item	A string whose value is the text of the item you want to add

Return value

Integer. Returns the position of the new item. If the list is sorted, the position returned is the position of the item after the list is sorted. Returns *-1* if it fails.

Usage

If the ListBox already contains items, AddItem adds the new item to the end of the list. If the list is sorted (its Sorted attribute is TRUE), PowerBuilder re-sorts the list after the item is added.

A list can have duplicate items. Items in the list are tracked by their position in the list, not their text.

> **Adding many items to a list with a horizontal scrollbar**
> If a ListBox or the ListBox portion of a DropDownListBox will have a large number of items, and you want to display an HScrollBar, call the SetRedraw function to turn Redraw off, add the items, call SetRedraw again to set Redraw on, and then set the HScrollBar attribute to TRUE. Otherwise, it may take longer than expected to add the items.
>
> **VBX controls**
> If you have created a VBX user object using a VBX control that supports the AddItem method, use the AddItem or InsertItem function instead of the AddItem method.

Example

This example adds the item Edit File to the ListBox lb_Actions:

```
integer rownbr
string s

s = "Edit File"
rownbr = lb_Actions.AddItem(s)
```

If lb_Actions contains Add and Run and the Sorted attribute is FALSE, the statement above returns 3 because Edit File becomes the third and last item. If the Sorted attribute is TRUE, the statement above returns 2 because Edit File becomes the second item after the list is sorted alphabetically.

See also

DeleteItem
FindItem
InsertItem
Reset
TotalItems

AddSeries

Description

Adds a series to a graph, naming it with the specified name. The new series is also assigned a number. A graph's series are numbered consecutively, according to the order they are added.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**AddSeries** (*seriesname*)

Parameter	Description
controlname	The name of the graph in which you want to add a series.
seriesname	A string whose value is the name of the series you want to add to *controlname*.

Return value

Integer. Returns the number assigned to the series if it succeeds. If *seriesname* is a duplicate, AddSeries returns the number of the existing series. If an error occurs, it returns *-1*.

Usage

Adds *seriesname* to the graph *controlname* and assigns the series a number. The number identifies the series within the graph. The numbers are assigned in sequence. The first series you add to the graph is assigned number 1 and is the first series displayed in the graph; the next is assigned 2, and so on.

The series name must be unique within the graph. If you specify a name that already exists in the graph, AddSeries returns the number of the existing series.

Series names are unique if they have different capitalization.

The series name can be an empty string (""). However, because series names must be unique, only one series can have a blank name.

If you want to insert a series in the middle of the list, use InsertSeries. You can also use InsertSeries to add a series to the end of the list, as AddSeries does, although it requires an additional argument to do it.

To add data to a series in the graph, use the AddData or InsertData function. To add a category to a series, use the InsertCategory or AddCategory function.

Examples

These statements add the series named Costs to the graph gr_product_data:

```
integer series_nbr
series_nbr = gr_product_data.AddSeries("Costs")
```

These statements add an unnamed series to the graph gr_product_data:

```
integer series_nbr
series_nbr = gr_product_data.AddSeries("")
```

See also

AddCategory
AddData
DeleteData
DeleteSeries
FindSeries
InsertCategory
InsertSeries
SeriesCount
SeriesName

ArrangeSheets

Description

Arranges the windows contained in an MDI frame. (Windows that are contained in an MDI frame are called sheets.) You can arrange the open sheets and the icons of minimized sheets or just the icons.

Applies to

MDI frame windows

Syntax

mdiframe.**ArrangeSheets** (*arrangetype*)

Parameter	Description
mdiframe	The name of an MDI frame window.

Parameter	Description
arrangetype	A value of the ArrangeTypes enumerated data type specifying how you want the open sheets arranged in the MDI frame window. Values are: ♦ Cascade!—Cascade the sheets that are not minimized, so that each sheet's title bar is visible. Also, arrange icons of minimized sheets in a row at the bottom of the frame. ♦ Layer! — Layer the sheets that are not minimized so that each sheet completely covers the one below it. Also, arrange icons of minimized sheets in a row at the bottom of the frame. ♦ Tile! — Tile the sheets that are not minimized so that they do not overlap. Also, arrange icons of minimized sheets in a row at the bottom of the frame. ♦ TileHorizontal! — Tile the sheets that are not minimized so that each is beside the other without overlapping. Also, arrange icons of minimized sheets in a row at the bottom of the frame. ♦ Icons! — Arrange the minimized sheets in a row at the bottom of the frame. On the Macintosh, Icons! has no effect since sheets cannot be iconized.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Examples

This statement in the script for the Clicked event for a MenuItem tiles the open sheets that are not minimized in the MDI frame window called MDI_User:

```
MDI_User.ArrangeSheets(Tile!)
```

This statement in the script for the Clicked event for a MenuItem arranges the icons of the minimized sheets at the bottom of the MDI frame window called MDI_User (not applicable to Macintosh):

```
MDI_User.ArrangeSheets(Icons!)
```

See also

GetActiveSheet
OpenSheet

Asc

Description

Converts the first character of a string to its ASCII integer value.

Syntax

Asc (*string*)

Parameter	Description
string	The string for which you want the ASCII value of the first character

Return value

Integer. Returns the ASCII value of the first character in *string*.

Usage

You can use Asc to find out the case of a character by testing whether its ASCII value is within the appropriate range.

Examples

This statement returns 65, the ASCII value for uppercase A:

```
Asc("A")
```

This example checks to see if the first character of string ls_name is uppercase:

```
String ls_name
IF Asc(ls_name) > 64 and Asc(ls_name) < 91 THEN ...
```

This example is a function that converts an array of integers into a string. Each integer specifies two characters. Its low byte is the first character in the pair and the high byte (ASCII * 256) is the second character. The function has an argument, iarr, which is the integer array:

```
string str_from_int, hold_str
integer arraylen
arraylen = UpperBound(iarr)

FOR i = 1 to arraylen
   // Convert first character of pair to a char
   hold_str = Char(iarr[i])

   // Add characters to string after converting
   // the integer's high byte to char
   str_from_int = &
      str_from_int + hold_str + &
      Char((iarr[i] - Asc(hold_str)) / 256)
NEXT
```

For sample code that builds the integer array from a string, see Mid.

See also

Char
Asc in Chapter 2, "DataWindow Painter Functions"

Beep

Description

Causes the computer to beep up to 10 times.

Syntax

Beep (*n*)

Parameter	Description
n	The number of times you want the computer to beep. If *n* is greater than 10, the computer beeps 10 times.

Return value

Integer. Returns *1* if it succeeds and *-1* if it fails. The return value usually is not used.

Example

This statement causes the computer to beep five times:

```
Beep(5)
```

Blob

Description

Converts a string to a blob data type.

Syntax

Blob (*text*)

Parameter	Description
text	The string you want to convert to a blob data type

Return value

Blob. Returns the converted string.

Example

This example saves a text string as a blob data type:

```
Blob B
B = Blob("Any Text")
```

BlobEdit

Description

Inserts data of any PowerBuilder data type into a blob variable.

Syntax

BlobEdit (*blobvariable, n, data*)

Parameter	Description
blobvariable	An initialized variable of the blob data type into which you want to copy a standard PowerBuilder data type
n	The number (1 to 4,294,967,295) of the position in *blobvariable* at which you want to begin copying the data
data	Data of a valid PowerBuilder data type that you want to copy into *blobvariable*

Return value

Unsigned long. Returns the position at which the next data can be copied if it succeeds, and NULL if there is not enough space in *blobvariable* to copy the data.

Examples

This code copies a bitmap in the blob emp_photo starting at position 1, stores the position at which the next copy can begin in nbr, and then copies a date into the blob emp_photo after the bitmap data:

```
blob{1000} emp_photo
blob temp
date pic_date
ulong nbr

... // Read BMP file containing employee picture
... // into temp using FileOpen and FileRead.
```

```
pic_date = Today( )
nbr = BlobEdit(emp_photo, 1, temp)
BlobEdit(emp_photo, nbr, pic_date)
UPDATEBLOB Employee SET pic = :emp_photo
   WHERE ...
```

See also

Blob
BlobMid

BlobMid

Description

Extracts data from a blob variable.

Syntax

BlobMid (*data, n* {, *length* })

Parameter	Description
data	Data of the blob data type
n	The number (1 to 4,294,967,295) of the first byte you want returned
length (optional)	The number of bytes (1 to 4,294,967,295) you want returned

Return value

Blob. Returns *length* bytes from *data* starting at byte *n*. If *n* is greater than the number of bytes in *data*, BlobMid returns an empty blob. If together *length* and *n* add up to more bytes than the blob contains, BlobMid returns the remaining bytes and the returned blob will be shorter than the specified length.

> **Tip**
> String variables contain a 0 terminator, which accounts for 1 byte. Include the terminator character when calculating how much data to extract.

Examples

These statements store 10 bytes of the blob datablob, starting at position 5, in data_1 and then store all the bytes of datablob, from position 5 to the end, in data_2:

```
blob data_1, data_2, datablob

... // Read a blob data type into datablob.
data_1 = BlobMid(datablob, 5, 10)
data_2 = BlobMid(datablob, 5)
```

This code copies a bitmap in the blob emp_photo starting at position 1, stores the position at which the next copy can begin in nbr, and then copies a date into the blob emp_photo after the bitmap data. Then, using the date's start position, it extracts the date from the blob and displays it in the StaticText st_1:

```
blob{1000} emp_photo
blob temp
date pic_date
ulong nbr

... // Read BMP file containing employee picture
... // into temp using FileOpen and FileRead.

pic_date = Today( )
nbr = BlobEdit(emp_photo, 1, temp)
BlobEdit(emp_photo, nbr, pic_date)
st_1.Text = String(Date(BlobMid(emp_photo, nbr)))
```

See also

Blob
BlobEdit

Cancel

Description

Stops the execution of a pipeline object.

Applies to

Pipeline objects

Syntax

pipelineobject.**Cancel** ( )

Parameter	Description
pipelineobject	The name of a pipeline user object that contains the pipeline object to be executed

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Call this function only when Start or Repair is executing.

When you stop a pipeline with Cancel, data is committed as if the pipeline reached the maximum errors limit. You control how the pipeline behaves when it reaches the limit in the Data Pipeline painter (see the *User's Guide*).

Example

This statement for a CommandButton's Clicked script allows the user to stop the execution of the pipeline i_pipe:

```
i_pipe.Cancel( )
```

See also

Repair
Start

CanUndo

Description

Tests whether Undo can reverse the most recent edit for an editable control.

Applies to

Any editable control (DataWindow, EditMask, MultiLineEdit, or SingleLineEdit)

Syntax

editname.**CanUndo** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, or SingleLineEdit for which you want to determine whether the last edit can be reversed by the Undo function. In a DataWindow, CanUndo applies to the edit control over the current row and column.

Return value

Boolean. Returns TRUE if the last edit can be reversed (undone) using the Undo function and FALSE if the last edit cannot be reversed.

Examples

These statements check to see if the last edit in mle_contact can be reversed; if yes the statements reverse it, and if no they display a message:

```
IF mle_contact.CanUndo( ) THEN
   mle_contact.Undo( )
ELSE
   MessageBox(Parent.Title, "Nothing to Undo")
END IF
```

See also Undo

CategoryCount

Description Counts the number of categories on the category axis of a graph.

Applies to Graph controls in windows and user objects, and graphs in DataWindow controls.

Syntax *controlname*.**CategoryCount** ({ *graphcontrol* })

Parameter	Description
controlname	The name of the graph for which you want the number of categories, or the name of DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow for which you want the number of categories. *Graphcontrol* is required if *controlname* is a DataWindow control.

Return value Integer. Returns the count if it succeeds and *-1* if an error occurs.

Examples These statements get the number of categories in the graph gr_revenues in the DataWindow control dw_findata:

```
integer li_count
li_count =  &
   dw_findata.CategoryCount("gr_revenues")
```

These statements get the number of categories in the graph gr_product_data:

```
integer li_count
li_count = gr_product_data.CategoryCount( )
```

See also

DataCount
SeriesCount

CategoryName

Description

Obtains the category name associated with the specified category number.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls.

Syntax

controlname.**CategoryName** ({ *graphcontrol*, } *categorynumber*)

Parameter	Description
controlname	The name of the graph in which you want to find the name of a specific category, or the name of the DataWindow *control* containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow for which you want the name of a specific category. *Graphcontrol* is required if *controlname* is a DataWindow control.
categorynumber	The number of the category for which you want the name.

Return value

String. Returns the name of *categorynumber* in *controlname*. If an error occurs, it returns the empty string ("").

Usage

Categories are numbered consecutively, from 1 to the value returned by CategoryCount. When you delete a category, the categories are renumbered to keep the numbering consecutive. You can use CategoryName to find out the named category associated with a category number.

Examples

These statements obtain the name of category 5 in the graph gr_product_data:

```
string ls_name
ls_name = gr_product_data.CategoryName(5)
```

These statements obtain the name of category 5 in the graph gr_revenues in the DataWindow control dw_findata:

```
string ls_name
ls_name = &
   dw_findata.CategoryName("gr_revenues", 5)
```

See also

AddCategory
SeriesName

Ceiling

Description

Determines the smallest whole number that is greater than or equal to a specified limit.

Syntax

Ceiling (*n*)

Parameter	Description
n	The number for which you want the smallest whole number that is greater than or equal to it

Return value

The data type of *n*. Returns the smallest whole number that is greater than or equal to *n*.

Examples

These statements set num to 5:

```
decimal dec, num
dec = 4.8
num = Ceiling(dec)
```

These statements set num to -4:

```
decimal num
num = Ceiling(-4.2)
num = Ceiling(-4.8)
```

See also

Int
Round
Truncate
Ceiling in Chapter 2, "DataWindow Painter Functions"

ChangeMenu

Description

Changes the menu associated with a window. If the window is an MDI frame window, ChangeMenu appends the list of open sheets to the currently active menu.

Applies to

Window objects

Syntax

windowname.**ChangeMenu** (*menuname* {, *position* })

Parameter	Description
windowname	The name of the window for which you want to change the menu.
menuname	The name of the menu you want to make the current menu for the window.
position (MDI frame windows only)	The number of the item on the menu bar to which you want to append the names of the open sheets. Items on the menu bar are numbered from the left, beginning with 1. The default is 1, which lists the open sheets on the menu bar's first menu.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

If you are changing the menu associated with an MDI frame window, the new menu will not be visible if an open sheet with its own menu is active. When a sheet has its own menu, the list of open sheets appears on its menu, as well as the hidden menu for the frame.

Example

This statement changes the top-level menu of the w_Employee window to m_Empl:

```
w_Employee.ChangeMenu(m_Empl)
```

Char

Description

Extracts the first character of a string or converts an integer to a char.

Syntax

Char (*n*)

Parameter	Description
n	A string that begins with the character you want, an integer you want to convert, or a blob in which the first value is a string or integer. The rest of the contents of the blob is ignored.

Return value

Char. Returns the first character of *n*.

Examples

This example sets ls_S to an asterisk, the character corresponding to the ASCII value 42:

```
string ls_S
ls_S = Char(42)
```

These statements generate delivery codes A to F for the values 1 through 6 of li_DeliveryNbr:

```
string ls_Delivery
integer li_DeliveryNbr

FOR li_DeliveryNbr = 1 to 6
   ls_Delivery = Char(64 + li_DeliveryNbr)
   ... // Additional processing of ls_Delivery
NEXT
```

See also

Asc
Char in Chapter 2, "DataWindow Painter Functions"

Check

Description

Displays a checkmark next to an item in a dropdown or cascading menu and sets the item's Checked attribute to TRUE.

Applies to

Menuitems in Menu objects

Syntax

menuitem.**Check** ( )

Parameter	Description
menuitem	The fully qualified name of the MenuItem next to which you want to display a checkmark. The item must be in a dropdown or cascading menu, not an item on a menu bar.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A checkmark next to a menu item indicates that the menu option is currently on and that the user can turn the option on and off by choosing it. For example, in the Window painter's Design menu, a checkmark is displayed next to Grid when the grid is on.

You can use Check in an item's Clicked script to mark a menu item when the user turns the option on and Uncheck to remove the check when the user turns the option off.

Equivalent syntax

You can set the MenuItem's Checked attribute instead of calling Check:

menuitem.Checked = TRUE

This statement:

```
Menu_Appl.M_View.M_Grid.Checked = TRUE
```

is equivalent to:

```
Menu_Appl.M_View.M_Grid.Check( )
```

Example

This statement displays a checkmark next to the MenuItem m_Grid in the m_View dropdown menu on the menu bar m_Appl:

```
m_Appl.m_View.m_Grid.Check( )
```

See also

Uncheck

ClassName

Description

Determines the class of an object or the data type of a variable.

Applies to

(Syntax 1) Any control

Syntax 2 is not an object function.

Syntax 1

controlname.**ClassName** ( )

Parameter	Description
controlname	The name of the control for which you want to know the name assigned to the control in the style window (the class of the control)

Return value 1

String. Returns the class of *controlname*, the name assigned to the control. Returns the empty string ("") if an error occurs.

Syntax 2

ClassName (*variable*)

Parameter	Description
variable	The name of the variable for which you want to know its name (that is, its data type)

Return value 2

String. Returns the name of *variable*. Returns the empty string ("") if an error occurs.

Usage

The class is the name of an object. You assign the name when you save the object in its painter. Usually, the class and the object itself appear to be the same because PowerBuilder declares a variable with the same name as the class for the object. However, if you have declared multiple instances of an object, it is clear that the object's class and the object's variable are different.

If an ancestor object has been instantiated with one of its descendants, you can use Syntax 1 of ClassName to find out the name of the descendant.

TypeOf reports an object's built-in object type. The types are values of the Object enumerated data type, such as Window! or CheckBox!. ClassName reports the class of the object in the ancestor-descendant hierarchy.

Examples

These statements return the class of the dragged control Source:

```
DragObject Source
string which_class

Source = DraggedObject( )
which_class = Source.ClassName( )
```

These statements return the class of the objects in the control array and store them in the_class array:

```
string the_class[]
windowobject the_object[]
integer i

FOR i = 1 TO UpperBound(control[])
   the_object[i] = control[i]
   the_class[i] = the_object[i].ClassName( )
NEXT
```

Suppose your object hierarchy has a window named ancestor_window and it has descendants called win1 and win2, and the user can choose which descendant to open as a sheet. The following code tests which descendant window class is currently active. The MDI frame is w_frame:

```
ancestor_window active_window
active_window = w_frame.GetActiveSheet()
IF ClassName(active_window) = "win1" THEN
   . . .
END IF
```

Syntax 2

If gd_doublenum is a global double variable, then ClassName sets varname to double:

```
string varname
varname = ClassName(gd_doublenum)
```

See also

DraggedObject
TypeOf

Clear

Description

Deletes selected text or an OLE 2.0 object from the specified control, but does not store it in the clipboard.

Applies to

MultiLineEdit, SingleLineEdit, DropDownListBox, OLE 2.0 controls, OLEStorage objects

Syntax

objectname.**Clear** ( )

Parameter	Description
objectname	The name of the EditMask, MultiLineEdit, or SingleLineEdit or DropDownListBox from which you want to delete (clear) selected text or the name of an OLE 2.0 control or storage object variable (type OLEStorage) from which you want to release its OLE object. If *objectname* is a DropDownListBox, its AllowEdit attribute must be TRUE.

Return value

Integer.

For edit controls, returns the number of characters that Clear removed from *objectname*. If no text is selected, no characters are removed and Clear returns *0*. If an error occurs, Clear returns *-1*.

For OLE 2.0 controls and storage variables, returns *0* if it succeeds and *-9* if an error occurs.

Usage

To select text for deleting, the user can use the mouse or keyboard. You can also call the SelectText function in a script.

To delete selected text and store it in the clipboard, use the Cut function.

Clearing the OLE object from an OLE 2.0 control deletes all references to it. Any changes to the object are not saved in its storage object or file.

Clearing an OLEStorage object variable breaks any connections established by Open or SaveAs between it and a storage file (when Open or SaveAs is called for the OLEStorage object variable). It also breaks connections between it and any OLE 2.0 controls that have called Open or SaveAs to connect to the object in the storage variable.

Examples

If the text in sle_comment1 is Draft and it is selected, this statement clears Draft from sle_comment1 and returns 5:

```
sle_comment1.Clear( )
```

If the text in sle_comment1 is Draft, the first statement selects the D and the second clears D from sle_comment1 and returns 1:

```
sle_comment1.SelectText(1,1)
sle_comment1.Clear( )
```

This example clears the object associated with the OLE 2.0 control ole_1, leaving the control empty:

```
integer result
result = ole_1.Clear( )
```

This example clears the object in the OLEStorage object variable olest_stuff. It also leaves any OLE 2.0 controls that have opened the object in olest_stuff empty too:

```
integer result
result = olest_stuff.Clear( )
```

See also

Close
Cut
Paste
ReplaceText
SelectText

ClearValues

Description

Deletes all the items from the list of values in a DropDownListBox associated with a DataWindow column. (This list of values is called a code table when it has both display and data values.) ClearValues does not affect the data stored in the column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ClearValues** (*column*)

Parameter	Description
datawindowname	The name of a DataWindow control or child DataWindow.
column	The column whose value list you want to delete. The column's edit style should be DropDownListBox. *Column* can be a column number (integer) or a column name (string).

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Example

This statement clears all values from the dropdown listbox of dw_Employee's status column:

```
dw_Employee.ClearValues("status")
```

See also

GetValue
SetValue

Clipboard

Description

Retrieves or replaces the contents of the system clipboard. There are two syntaxes:

- To retrieve or replace the contents of the system clipboard with text, use Syntax 1.
- To replace the contents of the system clipboard with a bitmap image of a graph, use Syntax 2.

Applies to

(Syntax 2) Graph controls in windows and user objects, and graphs in DataWindow controls. Syntax 1 is not an object function.

Syntax 1

Clipboard ({ *string* })

Parameter	Description
string (optional)	A string whose value is the text you want to place in the clipboard. The string replaces the current contents of the clipboard, if any.

Return value 1

String. Returns the current contents of the clipboard if the clipboard contains text. If *string* is specified, Clipboard returns the current contents and replaces it with *string*.

Returns the empty string ("") if the clipboard is empty or it contains nontext data, such as a bitmap. If *string* is specified, the nontext data is replaced with *string*.

Syntax 2

controlname.**Clipboard** ({ *graphcontrol* })

Parameter	Description
controlname	The name of the graph or the DataWindow control containing the graph you want to copy to the clipboard
graphcontrol (DataWindow control only)	A string whose value is the name of the graph control in the DataWindow object that you want to copy to the clipboard

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

You can use Syntax 1 of Clipboard with the Paste, Replace, or ReplaceText functions to insert the contents of the clipboard in an edit box or StaticText control.

Examples

These statements put the contents of the clipboard in the variable ls_CoName:

```
string ls_CoName
ls_CoName = Clipboard( )
```

The following statements place the contents of the clipboard in Heading, and then replace the contents of the clipboard with the string Employee Data:

```
string Heading
Heading = Clipboard("Employee Data")
```

The following statement replaces the selected text in the MultiLineEdit mle_terms with the contents of the clipboard:

```
mle_terms.ReplaceText(Clipboard( ))
```

The following statement exchanges the contents of the st_welcome with the contents of the clipboard:

```
st_welcome.Text = Clipboard(st_welcome.Text)
```

Syntax 2

This statement copies the graph gr_products_data to the clipboard:

```
gr_products_data.Clipboard( )
```

This statement copies the graph gr_employees in the DataWindow control dw_emp_data to the clipboard:

```
dw_emp_data.Clipboard("gr_employees")
```

See also

Clear
Copy
Cut
Paste
Replace
ReplaceText

Close

Description

Closes a window or an OLE storage or stream. There are several syntaxes:

- To close a window, use Syntax 1.
- To save the object in the specified OLEStorage object variable and clear any connections between it and a storage file or object, use Syntax 2. Close also severs connections with any OLE 2.0 controls that have opened the object. Calling Close is the same as calling Save and then Clear.
- To close the stream associated with the specified OLEStream object variable, use Syntax 3.

Applies to

(Syntax 1) Window objects

(Syntax 2) OLEStorage objects

(Syntax 3) OLEStream objects

Syntax 1

Close (*windowname*)

Parameter	Description
windowname	The name of the window you want to close

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Syntax 2

olestorage.**Close** ( )

Parameter	Description
olestorage	The OLEStorage object variable that you want to save and close

Return value 2

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The storage is not open
- *-9* Other error

Syntax 3

olestream.**Close** ( )

Parameter	Description
olestream	The OLEStream object variable that you want to close

Return value 3

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The stream is not open
- *-9* Other error

Usage

Use Syntax 1 of Close to close a window and release the storage occupied by the window and all the controls in the window.

When you call Close, PowerBuilder removes the window from view, closes it, executes the scripts for the CloseQuery and Close events (if any), and then executes the rest of the statements in the script that called the Close function.

After a window is closed, its attributes, instance variables, and controls can no longer be referenced in scripts. If a statement in the script references the closed window or its attributes or instance variables, an execution error will result.

Preventing a window from closing
You can prevent a window from being closed by setting Message.ReturnValue to 1 in the script for the CloseQuery event. You must make the setting in the last statement in the script.

Examples

These statements close the window w_employee and then open the window w_departments:

```
Close(w_employee)
Open(w_departments)
```

The following statements in the script for the CloseQuery event prompt the user for confirmation before they exit the application:

```
IF MessageBox('ExitApplication',  &
      'Exit?', Question!, YesNo!) = 1 THEN
   // If yes, close the window. If no, no action.
   Close(w_employee)
END IF
```

Syntax 2

This example saves and clears the object in the OLEStorage object variable olest_stuff. It also leaves any OLE 2.0 controls that have opened the object in olest_stuff empty too:

```
integer result
result = olest_stuff.Close( )
```

See also

Hide
Open
Save
SaveAs

CloseChannel

Description

Closes a DDE channel.

Platform information
This and other DDE functions have no effect on the Macintosh.

Syntax

CloseChannel (*handle* {, *windowhandle* })

Parameter	Description
handle	A long that identifies the DDE channel that will be closed. It is the same value returned by the OpenChannel function that opened the DDE channel.
windowhandle (optional)	The handle to the PowerBuilder window that is acting as the DDE client.

Return value

Integer. Returns *1* if it succeeds.

If an error occurs, CloseChannel returns a negative integer. Possible values are:

- *-1* Open failed
- *-2* The channel refuses to close
- *-3* No confirmation from the server
- *-9* *Handle* is NULL

Usage

Use CloseChannel to close a channel to a DDE server application that was opened by calling the OpenChannel function.

Although you can usually close the DDE channel by specifying just the channel's handle, it is a good idea to also specify the handle for PowerBuilder window associated with the channel. If you specify *windowhandle*, CloseChannel closes the DDE channel in the window identified by *windowhandle*. If you do not specify *windowhandle*, CloseChannel only closes the channel if it is associated with the active window. You can use the Handle function to obtain a window's handle.

Example

These statements open and close the channel identified by *handle*. The channel is associated with the window w_sheet:

```
long handle

handle = OpenChannel("Excel", "REGION.XLS", &
   Handle(w_sheet) )

... // Some processing

CloseChannel(handle, Handle(w_sheet))
```

See also

GetRemote
OpenChannel
SetRemote

CloseUserObject

Description

Closes a user object by removing it from view and executing the scripts for its Destructor event.

Applies to

Window objects

Syntax

windowname.**CloseUserObject** (*userobjectname*)

Parameter	Description
windowname	The name of the window that contains the user object
userobjectname	The name of the user object you want to close

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

Use CloseUserObject to close a user object and release the storage occupied by the object and its controls.

When you call CloseUserObject, PowerBuilder removes the object from view, closes it, executes the script for the Destructor event (if any), and then executes the rest of the statements in the script that called the CloseUserObject function.

After a user object is closed, its attributes, instance variables, and controls can no longer be referenced in scripts. If a statement in the script references the closed user object or its attributes or instance variables, an execution error will result.

Examples

These statements close the user object u_employee and then open the user object u_departments in the window w_personnel:

```
w_personnel.CloseUserObject(u_employee)
w_personnel.OpenUserObject(u_departments)
```

When the user chooses a menu item that closes a user object, the following excerpt from the menu item's script prompts the user for confirmation before closing the u_employee user object in the window to which the menu is attached:

```
IF MessageBox("Close ", "Close?",  &
      Question!, YesNo!) = 1 THEN

   // User chose Yes, close user object.
   ParentWindow.CloseUserObject(u_employee)
   // If user chose No, take no action.
END IF
```

See also OpenUserObject

CloseWithReturn

Description Closes a window and stores a return value in the Message object. You should only use CloseWithReturn for response windows.

Applies to Window objects

Syntax **CloseWithReturn** (*windowname*, *returnvalue*)

Parameter	Description
windowname	The name of the window you want to close
returnvalue	The value you want to store in the Message object when the window is closed. *Returnvalue* must be one of these data types: ♦ String ♦ Numeric ♦ PowerObject

Return value Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage The purpose of CloseWithReturn is to close a response window and return information from the response window to the window that opened it. Use CloseWithReturn to close a window, release the storage occupied by the window and all the controls in the window, and return a value.

Just as with Close, CloseWithReturn removes a window from view, closes it, and executes the script for the CloseQuery and Close events, if any. Before executing the event scripts, it also stores *returnvalue* in the Message object. Then PowerBuilder executes the rest of the script that called the Close function.

After a window is closed, its attributes, instance variables, and controls can no longer be referenced in scripts. If a statement in the script references the closed window or its attributes or instance variables, an execution error will result.

PowerBuilder stores *returnvalue* in the Message object attributes according to its data type. In the script that called CloseWithReturn, you can access the returned value by specifying the attribute of the Message object that corresponds to the return value's data type:

Return value data type	Message object attribute
Numeric	Message.DoubleParm
PowerObject (e.g., a structure)	Message.PowerObjectParm
String	Message.StringParm

Returning several values as a structure

To return several values, create a user-defined structure to hold the values and access the PowerObjectParm attribute of the Message object in the script that opened the response window. The structure is passed by value so you can access the information even if the original variable has been destroyed.

Referencing controls

User objects and controls are passed by reference, not by value. You cannot use CloseWithReturn to return a reference to a control on the closed window because the control no longer exists after the window is closed. Instead, return the value of one or more attributes of that control.

Preventing a window from closing

You can prevent a window from being closed by setting Message.ReturnValue to 1 in the script for the CloseQuery event. You must make the setting in the last statement in the script.

Examples

This statement closes the response window w_employee_response, returning the string emp_name to the window that opened it:

```
CloseWithReturn(Parent, "emp_name")
```

Suppose that a menu item opens one window if the user is a novice and another window if the user is experienced. The menu item displays a response window called w_signon to prompt the user for his or her experience level. The user types an experience level in the SingleLineEdit control w_signon_id. The OK button in the response window passes the text in w_signon_id back to the menu item script. The menu item script checks the StringParm attribute of the Message object and opens the desired window.

The script for the Clicked event of the OK button in the w_signon response window is a single line:

```
CloseWithReturn(Parent, sle_signon_id.Text)
```

The script for the MenuItem is:

```
string ls_userlevel

// Open the response window
Open(w_signon)

// Check text returned in Message object
ls_userlevel = Message.StringParm

IF ls_userlevel = "Novice" THEN
   Open(win_novice)
ELSE
   Open(win_advanced)
END IF
```

See also

Close
OpenSheetWithParm
OpenUserObjectWithParm
OpenWithParm

CommandParm

Description

Retrieves the parameter string, if any, that followed the program name when the application was executed.

Platform information
CommandParm has no effect on the Macintosh and returns the empty string.

Syntax

CommandParm ( )

Return value

String. Returns the application's parameter string if it succeeds and the empty string ("") if it fails or if there were no parameters.

Usage

The command parameters can follow the program name in the command line of a Windows program item or in the Program Manager's Run response window. For example, when the user chooses File**Error! AutoText entry not defined.**Run in the Program Manager and enters:

```
MyAppl C:\EMPLOYEE\EMPLIST.TXT
```

CommandParm retrieves the string C:\EMPLOYEE\EMPLIST.TXT.

If the application's command line includes several parameters, CommandParm returns them all as a single string. You can use string functions, such as Mid and Pos, to parse the string.

Examples

These statements in the script for the Open event for the application retrieve the command line parameters and save them in the variable ls_command_line:

```
string ls_command_line
ls_command_line = CommandParm( )
```

If the command line holds several parameters, you can use string functions to separate the parameters. The following example stores a variable number of parameters, obtained with CommandParm, in an array. The code assumes each parameter is separated by one space.

For each parameter, the Pos function searches for a space; the Left function copies the parameter to the array; and Replace removes the parameter from the original string so the next parameter moves to the first position:

```
string ls_cmd, ls_arg[]
integer i, li_argcnt

// Get the parameters and strip blanks
// from start and end of string
ls_cmd = Trim(CommandParm( ))

li_argcnt = 1
```

```
DO WHILE Len(ls_cmd ) > 0
   // Find the first blank
   i = Pos( ls_cmd, " " )
   // If no blanks (only one argument),
   // set i to point to the hypothetical character
   // after the end of the string
   if i = 0 then i = Len(ls_cmd) + 1

   // Assign the arg to the argument array.
   // Number of chars copied is one less than the
   // position of the space found with Pos
   ls_arg[li_argcnt] = Left( ls_cmd, i - 1 )

   // Increment the argument count for the next loop
   li_argcnt = li_argcnt + 1

   // Remove the argument from the string
   // so the next argument becomes first
   ls_cmd = Replace( ls_cmd, 1, i, "" )
LOOP
```

ConnectToNewObject

Description

Creates a new OLE object in the specified server application and associates it with a PowerBuilder OLEObject variable. ConnectToNewObject starts the server application, if necessary.

Applies to

OLEObject objects

Syntax

oleobject.**ConnectToNewObject** (*classname*)

Parameter	Description
oleobject	The name of an OLEObject variable which you want to connect to an OLE object. You cannot specify an OLEObject that is the Object attribute of an OLE 2.0 control.
classname	A string whose value is the name of an OLE class, which identifies an OLE server application and a type of object that the server can manipulate via OLE.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Invalid call: the argument is the Object attribute of a control
- *-2* Class name not found
- *-3* Object could not be created
- *-4* Could not connect to object
- *-9* Other error

Usage

The OLEObject variable is used for OLE automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically.

For more information about OLE automation, see ConnectToObject.

Example

This example creates an OLEObject variable and calls ConnectToNewObject to create a new Excel object and connect to it:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToNewObject(  &
   "excel.application")
```

See also

ConnectToObject
DisconnectObject

ConnectToObject

Description

Associates an OLE object with a PowerBuilder OLEObject variable and starts the server application. The OLEObject variable and ConnectToObject are used for OLE automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically.

Applies to

OLEObject objects

Syntax

oleobject.**ConnectToObject** (*filename* {, *classname* })

Parameter	Description
oleobject	The name of an OLEObject variable which you want to connect to an OLE object. You cannot specify an OLEObject that is the Object attribute of an OLE 2.0 control.
filename	A string whose value is the name of an OLE storage file. You can specify the empty string for *filename*, in which case you must specify *classname*. *Oleobject* is connected to the active object in the server application specified in *classname*.
classname (optional)	A string whose value is the name of an OLE class, which identifies an OLE server application and a type of object that the server can manipulate via OLE. If you omit *classname*, PowerBuilder uses the extension of *filename* to determine what server application to start.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Invalid call: the argument is the Object attribute of a control
- *-2* Class name not found
- *-3* Object could not be created
- *-4* Could not connect to object
- *-5* Can't connect to the currently active object
- *-6* Filename is not valid
- *-7* File not found or file couldn't be opened
- *-8* Load from file not supported by server
- *-9* Other error

Usage

After you have created an OLEObject variable and connected it to an OLE object and its server application, you can set attributes and call functions supported by the OLE server. PowerBuilder's compiler will not check the syntax of functions that you call for an OLEObject variable. If the functions are not present when the application is run or the attribute names are invalid, a runtime error occurs.

Declare and create an OLEObject variable
You must use the CREATE statement to allocate memory for an OLEObject variable, as shown in the example below.

When you create an OLEObject variable, make sure you destroy the object before it goes out of scope. When the object is destroyed it is disconnected from the server and the server is closed. If the object goes out of scope without disconnecting, there will be no way to halt the server application.

Check the documentation for the server application to find out what attributes and functions it supports. Some applications support a large number. For example, Excel has approximately 4000 operations you can automate.

The OLEObject data type supports OLE automation as a background activity in your application. You can also invoke server functions and attributes for an OLE object in an OLE 2.0 control. To do so, specify the Object attribute of the control before the server function name. When you want to automate an object in a control, you do not need an OLEObject variable.

For example, the following changes a value in an Excel cell for the object in the OLE 2.0 control ole_1:

```
ole_1.Object.application.cells(1,1).value = 14
```

Example

This example declares and creates an OLEObject variable and connects to an Excel worksheet, which is opened in Excel. It then sets a value in the worksheet, saves it, and destroys the OLEObject variable, which exits the Excel:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToObject(  &
   "c:\excel\expense.xls")
IF result = 0 THEN
   myoleobject.application.cells(1,1).value = 14
   myoleobject.application.save (???)
END IF
DESTROY myoleobject
```

This example connects to an Excel chart:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToObject(  &
   "c:\excel\expense.xls", "excel.chart")
```

This example connects to the currently active object in Excel, which is already running:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToObject("",  &
   "excel.application")
```

See also

ConnectToNewObject
DisconnectObject

Copy

Description

Puts selected text or an OLE object in the specified control on the clipboard. Copy does not change the source text or object.

Applies to

DataWindow, MultiLineEdit, SingleLineEdit, DropDownListBox, OLE 2.0 controls

Syntax

controlname.**Copy** ( )

Parameter	Description
controlname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox or OLE 2.0 control containing the text or OLE object you want to copy to the clipboard. If *controlname* is a DataWindow, text is copied from the edit box over the current row and column. If *controlname* is a DropDownListBox, its AllowEdit attribute must be TRUE.

Return value

Integer.

For edit controls, returns the number of characters that were copied to the clipboard. If no text is selected in *controlname*, no characters are copied and Copy returns *0*. If an error occurs, Copy returns *-1*.

For OLE 2.0 controls, returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Control is empty
- *-2* Copy failed
- *-9* Other error

Usage

To select text for copying, the user can use the mouse or keyboard. You can also call the SelectText function in a script.

To insert the contents of the clipboard into a control, use the Paste function.

Copy does not delete the selected text or OLE object. To delete the data, use the Clear or Cut function.

Examples

Assuming the selected text in mle_emp_address is Temporary Address, these statements copy Temporary Address from mle_emp_address to the clipboard and store 17 in copy_amt:

```
integer copy_amt
copy_amt = mle_emp_address.Copy( )
```

This example copies the OLE object in the OLE 2.0 control ole_1 to the clipboard:

```
integer result
result = ole_1.Copy( )
```

See also

Clear
Clipboard
Cut
Paste
ReplaceText
SelectText

Cos

Description

Calculates the cosine of an angle.

Syntax

Cos (*n*)

Parameter	Description
n	The angle (in radians) for which you want the cosine

Return value

Double. Returns the cosine of *n*.

Examples

This statement returns 1:

```
Cos(0)
```

This statement returns .540302:

```
Cos(1)
```

This statement returns -1:

```
Cos(Pi(1))
```

See also

Pi
Sin
Tan
Cos in Chapter 2, "DataWindow Painter Functions"

Cpu

Description

Reports the amount of CPU time that has elapsed since the application started.

Syntax

Cpu ( )

Return value

Long. Returns the number of milliseconds of CPU time elapsed since the start of your PowerBuilder application.

Example

These statements determine the amount of CPU time that elapsed while a group of statements executed:

```
// Declare ll_start and ll_used as long integers.
long ll_start, ll_used

// Set the start equal to the current CPU usage.
ll_start = Cpu( )
...  // Executable statements being timed
// Set used to the number of CPU seconds
// that were used (current CPU time - start).
ll_used = CPU( ) - ll_start
```

Create

Description

Creates a DataWindow object using DataWindow source code and puts that object in the specified DataWindow control. This "dynamic" DataWindow object does not become a permanent part of the application source library.

Applies to

DataWindow controls

Syntax

datawindowname.**Create** (*syntax* {, *errorbuffer* })

Parameter	Description
datawindowname	The name of the DataWindow control in which PowerBuilder will create the new DataWindow object.
syntax	A string whose value is the DataWindow source code that will be used to create the DataWindow object.
errorbuffer (optional)	The name of a string that will hold any error messages that occur. If you do not specify an error buffer, a message box will display the error messages.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The Create function creates a DataWindow object using the source code in *syntax*. It substitutes the new DataWindow object for the DataWindow object currently associated with *datawindowname*.

DataWindow source code syntax is quite complex. You can use the Describe and LibraryExport functions to obtain the source code of existing DataWindows to use as models. Another source of DataWindow code is the SyntaxFromSQL function, which creates DataWindow source code based on an SQL statement.

> **Tip**
> You can call SyntaxFromSQL directly as the value for *syntax*. However, this does not give you the chance to check the whether errors have been reported in its error argument. Before you use SyntaxFromSQL in Create make sure the SQL syntax is valid.

For details on DataWindow source code syntax, see the Appendixes of this manual, particularly Appendix B, "DataWindow Syntax Listing."

Example

These statements create a new DataWindow in the control dw_new from the DataWindow source code returned by the SyntaxFromSQL function. Errors from SyntaxFromSQL and Create are displayed in the MultiLineEdits mle_sfs and mle_create. After creating the DataWindow, you must call SetTransObject for the new DataWindow object before you can retrieve data:

```
string error_syntaxfromSQL, error_create
string new_sql, new_syntax

new_sql = 'SELECT emp_data.emp_id, '  &
   + 'emp_data.emp_name '  &
   + 'from emp_data '  &
   + 'WHERE emp_data.emp_salary>45000'

new_syntax = SQLCA.SyntaxFromSQL(new_sql,  &
   'Style(Type=Form)', error_syntaxfromSQL)

IF Len(error_syntaxfromSQL) > 0 THEN
   // Display errors
   mle_sfs.Text = error_syntaxfromSQL
ELSE
   // Generate new DataWindow
   dw_new.Create(new_syntax, error_create)
   IF Len(error_create) > 0 THEN
      mle_create.Text = error_create
   END IF
END IF

dw_new.SetTransObject(SQLCA)
dw_new.Retrieve()
```

See also

SyntaxFromSQL
SetTrans
SetTransObject

CrosstabDialog

Description

Displays the Crosstab Definition dialog so the user can modify the definition of a crosstab DataWindow during execution. The dialog is the same one that you use in the DataWindow painter to define the crosstab.

Applies to

DataWindow controls

Syntax

datawindowname.**CrosstabDialog** ( )

Parameter	Description
datawindowname	The name of the DataWindow control for which you want to display the Crosstab Definition dialog.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If the style of the DataWindow object in the DataWindow control is not crosstab, CrosstabDialog has no effect.

Example

This statement in the script for the CommandButton cb_define displays the Crosstab Definition dialog so the user can modify the definition of the crosstab DataWindow object in dw_1:

```
dw_1.CrosstabDialog( )
```

Cut

Description

Deletes selected text or an OLE object from the specified control and stores it on the clipboard. Cut replaces the clipboard contents, if any, with the deleted text or object.

Applies to

DataWindow, MultiLineEdit, SingleLineEdit, DropDownListBox, OLE 2.0 controls

Syntax

controlname.**Cut** ( )

Parameter	Description
controlname	The name of the DataWindow, MultiLineEdit, SingleLineEdit, DropDownListBox or OLE 2.0 control containing the text or object to be cut. If *controlname* is a DataWindow, text is cut from the edit box over the current row and column. If *controlname* is a DropDownListBox, the AllowEdit attribute must be TRUE.

Return value

Integer.

For editable controls, returns the number of characters that were cut from *controlname* and stored in the clipboard. If no text is selected, no characters are cut and Cut returns *0*. If an error occurs, Cut returns *-1*.

For OLE 2.0 controls, returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Control is empty
- *-2* Cut failed
- *-9* Other error

Usage

To select text for deleting, the user can use the mouse or keyboard. You can also call the SelectText function in a script.

To insert the contents of the clipboard into a control, use the Paste function.

To delete selected text or an OLE object but not store it in the clipboard, use the Clear function.

Cutting an OLE object breaks any connections between it and its source file or storage, just as Clear does. (See Clear.)

Examples

If the selected text in mle_emp_address is Temporary, this statement deletes Temporary from mle_emp_address, stores it in the clipboard, and returns 9:

```
mle_emp_address.Cut( )
```

This example cuts the OLE object in the OLE 2.0 control ole_1 and puts it on the clipboard:

```
integer result
result = ole_1.Cut( )
```

See also

Copy
Clear
Clipboard
DeleteItem
Paste

DataCount

Description

Reports the number of data points in the specified series in a graph.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**DataCount** ({ *graphcontrol*, } *seriesname*)

Parameter	Description
controlname	The name of the graph in which you want the number of data points in a specific series, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	The name of the graph in the DataWindow control for which you want the data point count for the series
seriesname	A string whose value is the name of the series for which you want the number of data points

Return value

Long. Returns the number of data points in the specified series if it succeeds and *-1* if an error occurs.

Examples

These statements store in ll_count the number of data points in the series named Costs in the graph gr_product_data:

```
long ll_count
ll_count = gr_product_data.DataCount("Costs")
```

These statements store in ll_count the number of data points in the series named Salary in the graph gr_dept in the DataWindow control dw_employees:

```
long ll_count
ll_count =  &
   dw_employees.DataCount("gr_dept", "Salary")
```

See also

AddSeries
InsertSeries
SeriesCount

Date

Description

Converts DateTime, string, or numeric data to data of type date. It also extracts a date value from a blob. You can use one of three syntaxes, depending on the data type of the source data:

- To extract the date from DateTime data, or to extract a date stored in a blob, use Syntax 1.
- To convert a string to a date, use Syntax 2.
- To combine numeric data into a date, use Syntax 3.

Syntax 1

Date (*datetime*)

Parameter	Description
datetime	A DateTime value or a blob in which the first value is a date or DateTime value. The rest of the contents of the blob is ignored.

Return value 1

Date. Returns the date in *datetime* as a date. If *datetime* does not contain a valid date, Date returns *1900-01-01*.

Syntax 2

Date (*string*)

Parameter	Description
string	A string whose value is a valid date (such as January 1, 1998, or 12-31-99) that you want returned as a date

Return value 2

Date. Returns the date in *string* as a date. If string does not contain a valid date, Date returns *1900-01-01*.

Syntax 3

Date (*year, month, day*)

Parameter	Description
year	The 4-digit year (-9999 to 9999) of the date
month	The 1- or 2-digit integer for the month (1 to 12) of the year
day	The 1- or 2-digit integer for the day (1 to 31) of the month

Return value 3

Date. Returns the date specified by the integers for *year*, *month*, and *day*, as a date data type. If any value is invalid (that is, out of the range of values for dates), Date returns *1900-01-01*.

Usage

Valid dates in strings can include any combination of day (1 to 31), month (1 to 12 or the name or abbreviation of a month), and year (2 or 4 digits). Leading zeros are optional for month and day. If the month is a name or an abbreviation, it can come before or after the day; if it is a number, it must be in the month location specified in the Windows control panel. PowerBuilder assumes a 4-digit number is a year.

Date literals, which do not need to be converted with the Date function, have the format yyyy-mm-dd.

Examples

After a value for the DateTime variable ldt_StartDateTime has been retrieved from the database, this example sets ld_StartDate equal to the date in ldt_StartDateTime:

```
DateTime ldt_StartDateTime
date ld_StartDate

ld_StartDate = Date(ldt_StartDateTime)
```

Suppose that the value of a blob variable ib_blob contains a DateTime value beginning at byte 32. The following statement converts it to a date value.

```
date ld_date
ld_date = Date(BlobMid(ib_blob, 32))
```

Syntax 2

These statements all return the date data type for text expressing the date July 4, 1994 (1994-07-04). The system setting for the month location is in the center:

```
Date("1994/07/04")

Date("1994 July 4")

Date("July 4, 1994")
```

The following groups of statements check to be sure the date in sle_start_date is a valid date and display a message if it is not. The first version checks the result of the Date function to see if the date was valid. The second uses the IsDate function to check the text before using Date to convert it.

Version 1:

```
// Windows Control Panel date format is YY/MM/DD
date ld_my_date

ld_my_date = Date( sle_start_date.Text )
IF ld_my_date = Date("1900-01-01") THEN
   MessageBox("Error", "This date is invalid: "  &
   + sle_start_date.Text)
END IF
```

Version 2:

```
date ld_my_date

IF IsDate(sle_start_date.Text) THEN
   ld_my_date = Date(sle_start_date.Text)
Else
   MessageBox("Error", "This date is invalid: "  &
   + sle_start_date.Text)
END IF
```

Syntax 3

These statements use integer values to set ld_my_date to 1994-10-15:

```
date ld_my_date
ld_my_date = Date(1994, 10, 15)
```

See also

DateTime
IsDate
Date in Chapter 2, "DataWindow Painter Functions"

DateTime

Description

Combines a date and a time value into a DateTime value (Syntax 1) or obtains a DateTime value that is stored in a blob (Syntax 2).

Syntax 1

DateTime (*date* {, *time* })

Parameter	Description
date	A value of type date.
time (optional)	A value of type time. If you omit *time*, PowerBuilder sets *time* to 00:00:00.000000 (midnight). If you specify *time*, only the hour portion is required.

Return value 1

DateTime. Returns a DateTime value based on the values in *date* and optionally *time*.

Syntax 2

DateTime (*blob*)

Parameter	Description
blob	A blob in which the first value is a DateTime value. The rest of the contents of the blob is ignored.

Return value 2

DateTime. Returns the DateTime value stored in *blob*.

Usage

DateTime data is used only for reading and writing DateTime values to and from a database. To use the date and time values in scripts, use the Date and Time functions to assign values to date and time variables.

Examples

These statements convert the date and time stored in ld_OrderDate and lt_OrderTime to a DateTime value that can be used to update the database:

```
DateTime ldt_OrderDateTime
date ld_OrderDate
time lt_OrderTime

ld_OrderDate = Date(sle_orderdate.Text)
lt_OrderTime = Time(sle_ordertime.Text)
ldt_OrderDateTime = DateTime(  &
   ld_OrderDate, lt_OrderTime)
```

Syntax 2

After assigning blob data from the database to lb_blob, the following example obtains the DateTime value stored at position 20 in the blob. (The length you specify for BlobMid must be at least as long as the DateTime value but can be longer):

```
DateTime dt
dt = DateTime(BlobMid(lb_blob, 20, 40))
```

See also

Date
Time
DateTime in Chapter 2, "DataWindow Painter Functions"

Day

Description

Obtains the day of the month in a date value.

Syntax

Day (*date*)

Parameter	Description
date	A date value from which you want the day

Return value

Integer. Returns an integer (1 to 31) representing the day of the month in *date*.

Examples

These statements extract the day (31) from the date literal 1994-01-31 and set li_day_portion to that value:

```
integer li_day_portion
li_day_portion = Day(1994-01-31)
```

These statements check to be sure the date in sle_date is valid and, if so, set li_day_portion to the day in the sle_date:

```
integer li_day_portion

IF IsDate(sle_date.Text) THEN
   li_day_portion = Day(Date(sle_date.Text))
ELSE
   MessageBox("Error",  &
      "This date is invalid: "  &
      + sle_date.Text)
END IF
```

See also

Date
IsDate
Month
Year
Day in Chapter 2, "DataWindow Painter Functions"

DayName

Description

Determines the day of the week in a date value and returns the weekday's name.

Syntax

DayName (*date*)

Parameter	Description
date	A date value for which you want the name of the day

Return value

String. Returns a string whose value is the weekday (Sunday, Monday, and so on) of *date*.

Examples

These statements evaluate the date literal 1993-07-04 and set day_name to Sunday:

```
string  day_name
day_name = DayName(1993-07-04)
```

These statements check to be sure the date in sle_date is valid and, if so, set day_name to the day in the sle_date:

```
string day_name

IF IsDate(sle_date.Text) THEN
   day_name = DayName(Date(sle_date.Text))
ELSE
   MessageBox("Error",  &
      "This date is invalid: "  &
      + sle_date.Text)
END IF
```

See also

Day
DayNumber
IsDate
DayName in Chapter 2, "DataWindow Painter Functions"

DayNumber

Description

Determines the day of the week of a date value and returns the number of the weekday.

Syntax

DayNumber (*date*)

Parameter	Description
date	The date value from which you want the number of the day of the week

Return value

Integer. Returns an integer (1-7) representing the day of the week of *date*. Sunday is day 1, Monday is day 2, and so on.

Examples

These statements evaluate the date literal 1990-01-31 and set day_nbr to 4 (January 31, 1990, was a Wednesday):

```
integer day_nbr
day_nbr = DayNumber(1990-01-31)
```

These statements check to be sure the date in sle_date is valid and, if so, set day_nbr to the number of the day in the sle_date:

```
integer day_nbr

IF IsDate(sle_date.Text) THEN
   day_nbr = DayNumber(Date(sle_date.Text))
ELSE
   MessageBox("Error",  &
      "This date is invalid: "  &
      + sle_date.Text)
END IF
```

See also

Day
DayName
IsDate
DayNumber in Chapter 2, "DataWindow Painter Functions"

DaysAfter

Description

Determines the number of days one date occurs after another.

Syntax

DaysAfter (*date1*, *date2*)

Parameter	Description
date1	A date value that is the start date of the interval being measured
date2	A date value that is the end date of the interval

Return value

Long. Returns a long whose value is the number of days *date2* occurs after *date1*. If *date2* occurs before *date1*, DaysAfter returns a negative number.

Examples

This statement returns 4:

```
DaysAfter(1995-12-20, 1995-12-24)
```

This statement returns -4:

```
DaysAfter(1995-12-24, 1995-12-20)
```

This statement returns 0:

```
DaysAfter(1995-12-24, 1995-12-24)
```

This statement returns 5:

```
DaysAfter(1994-12-29, 1995-01-03)
```

If you declare date1 and date2 date variables and assign February 16, 1995, to date1 and April 28, 1995, to date2 as shown below:

```
date date1, date2

date1 = 1995-02-16
date2 = 1995-04-28
```

then each of the following statements returns 71:

```
DaysAfter(date1, date2)
DaysAfter(1995-02-16, date2)
DaysAfter(date1, 1995-04-28)
DaysAfter(1995-02-16, 1995-04-28)
```

See also

SecondsAfter
DaysAfter in Chapter 2, "DataWindow Painter Functions"

DBCancel

Description

Cancels the retrieval in process in a data window. You must call DBCancel in the script for the RetrieveRow event in order to successfully interrupt the retrieval.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**DBCancel** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to cancel the retrieval in progress

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

To enable the user to cancel a retrieval, you need to provide a way for the user to signal the cancellation and then have code in the RetrieveRow event respond to that signal. In the example, the user clicks a button that sets a global variable, which the RetrieveRow event script checks after each row is retrieved.

The Clicked event script for the CommandButton cb_cancel sets the global boolean variable gb_cancel to TRUE when the user clicks the button:

```
// Signal retrieval cancellation
gb_cancel = TRUE
```

The script for the RetrieveRow event checks the gb_cancel variable and cancels the retrieval in progress in dw_employee if the variable has been set to TRUE:

```
If gb_cancel then dw_employee.DBCancel()
```

See also

Retrieve

DBErrorCode

Description

Reports the database-specific error code that triggered the DBError event.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**DBErrorCode** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the error number (code) of the database error that occurred

Return value

Long. Returns a database-specific error code generated by a database error in *datawindowname*. Returns *0* if there is no error.

Usage

When a database error occurs while a DataWindow control is interacting with the database, PowerBuilder triggers the DBError event. Since DBErrorCode is meaningful only if a database error has occurred, you should call the function only in the DBError event.

Examples

This statement returns the error code for dw_employee:

```
dw_employee.DBErrorCode( )
```

Since this function is meaningful only in a DataWindow DBError event, you can use the pronoun This instead of the DataWindow's name:

```
This.DBErrorCode( )
```

These statements check the error code for dw_employee and if it is -4, perform some processing:

```
long ll_Error_Nbr
ll_Error_Nbr = This.DBErrorCode( )
IF ll_Error_Nbr = -4 THEN ...
```

When an error occurs in dw_Emp, the following statements in the DBError event's script will display the error number and message. Calling SetActionCode suppresses the default error message:

```
long ll_Error_Nbr

ll_Error_Nbr = This.DBErrorCode( )
IF ll_Error_Nbr <> 0 THEN
   MessageBox("Database Error", "Number "  &
      + string(ll_Error_Nbr) + " "  &
      + This.DBErrorMessage( ), StopSign!)

   // Stop PowerBuilder from displaying the error
   This.SetActionCode(1)
END IF
```

See also

DBErrorMessage
MessageBox

DBErrorMessage

Description

Reports the database-specific error message that triggered the DBError event.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**DBErrorMessage** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the database error message

Return value

String. Returns a string whose value is a database-specific error message generated by a database error in *datawindowname*. Returns the empty string ("") if there is no error.

Usage

When a database error occurs while a DataWindow control is interacting with the database, PowerBuilder triggers the DBError event. Since DBErrorCode is meaningful only if a database error has occurred, you should call the function only in the DBError event.

Examples

This statement returns the error message generated by a database error in dw_employee:

```
dw_employee.DBErrorMessage( )
```

Since this function is meaningful only in a DataWindow, you can use the pronoun This instead of the DataWindow's name:

```
This.DBErrorCode( )
```

If data processing fails in dw_Emp and these statements are coded in the script for the DBError event, a message box containing the error number and the message displays:

```
string err_msg

err_msg = This.DBErrorMessage( )
IF err_msg <> "" THEN
   MessageBox("DBError", "Number" +  &
      String(This.DBErrorCode( ))+ " " +  &
      err_msg, StopSign!)

   // Stop PowerBuilder from displaying the error
   This.SetActionCode(1)
END IF
```

See also

DBErrorCode
MessageBox

DBHandle

Description

Reports the handle for your DBMS.

Applies to

Transaction objects

Syntax

transactionobject.**DBHandle** ( )

Parameter	Description
transactionobject	The current transaction object

Return value

Long. Returns the handle for your DBMS. *Transactionobject* must exist, and the database must be connected. If *transactionobject* exists but is not connected, DBHandle returns *0*. If *transactionobject* does not exist, an execution error occurs.

Usage

PowerBuilder uses the database handle internally to communicate with the database. If your database supports an API with functions that PowerBuilder doesn't support, you can use DBHandle to provide the handle as an argument to one of these external functions.

Examples

For examples, search for "calling external database functions" in online Help.

Dec

Description

Converts a string to a decimal number or obtains a decimal value stored in a blob.

Syntax

Dec (*stringorblob*)

Parameter	Description
stringorblob	The string whose value you want returned as a decimal value or a blob in which the first value is the decimal you want. The rest of the contents of the blob is ignored.

Return value

Decimal. Returns the value of *stringorblob* as a decimal. If *stringorblob* is not a valid PowerScript number, Dec returns *0*.

Examples

This statement returns 24.3 as a decimal data type:

```
Dec("24.3")
```

This statement returns the contents of the SingleLineEdit sle_salary as a decimal number:

```
Dec(sle_salary.Text)
```

For an example of assigning and extracting values from a blob, see Real.

See also

Double
Integer
Long
Real

DeleteCategory

Description

Deletes a category and the data values for that category from the category axis of a graph.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**DeleteCategory** (*categoryvalue*)

Parameter	Description
controlname	The graph in which you want to delete a category.
categoryvalue	A value that is the category you want to delete from *controlname*. The value you specify must be the same data type as the data type of the category axis.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

These statements delete the category whose name is entered in the SingleLineEdit sle_delete from the graph gr_product_data:

```
string CategName
CategName = sle_delete.Text
gr_product_data.DeleteCategory(CategName)
```

See also

DeleteData
DeleteSeries

DeleteData

Description

Deletes a data point from a series of a graph. The remaining data points in the series are shifted left to fill the data point's category.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**DeleteData** (*seriesnumber*, *datapointnumber*)

Parameter	Description
controlname	The name of the graph in which you want to delete a data value
seriesnumber	The number of the series containing the data value you want to delete from *controlname*
datapointnumber	The number of the data point containing the data you want to delete

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

These statements delete the data at data point 7 in the series named Costs in the graph gr_product_data:

```
integer SeriesNbr
// Get the number of the series.
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.DeleteData(SeriesNbr, 7)
```

See also

AddData
DeleteCategory
DeleteSeries
FindSeries

DeletedCount

Description

Reports the number of rows that have been marked for deletion in the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**DeletedCount** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of rows that have been deleted, but not updated, in the associated database table

Return value

Long. Returns the number of rows that have been deleted from *datawindowname* but not updated in the associated database table.

Returns *0* if no rows have been deleted or all the deleted rows have been updated in the database table. Returns *-1* if it fails.

Usage

An updatable DataWindow control has several buffers. The primary buffer stores the rows currently being displayed. The delete buffer stores rows that the application has marked for deletion by calling the DeleteRow function. These rows are saved until the database is updated. You can use DeletedCount to find out if there are any rows in the delete buffer.

Examples

If two rows in dw_employee have been deleted but have not been updated in the associated database table, then these statements set ll_Del to 2:

```
Long ll_Del
ll_Del = dw_employee.DeletedCount( )
```

The following example tests whether there are rows in the delete buffer and, if so, updates the database table associated with dw_employee:

```
Long ll_Del
ll_Del = dw_employee.DeletedCount( )
If ll_Del <> 0 then dw_employee.Update( )
```

See also

DeleteRow
FilteredCount
ModifiedCount
RowCount
Update

DeleteItem

Description

Deletes an item from the list of values for a ListBox or DropDownListBox control.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**DeleteItem** (*index*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want to delete an item
index	The position number of the item you want to delete

Return value

Integer. Returns the number of items remaining in the list of values after the item is deleted. If an error occurs, DeleteItem returns *-1*.

Usage

If the control's Sorted attribute is set, the order of the list is probably different from the order you specified when you defined the control. If you know the item's text, use FindItem to determine the item's index.

VBX controls
If you have created a VBX user object using a VBX control that supports the RemoveItem method, use the DeleteItem function call instead of the RemoveItem method.

Examples

If lb_actions contains 10 items, this statement deletes item 5 from lb_actions and returns 9:

```
lb_actions.DeleteItem(5)
```

These statements delete the first selected item in lb_actions:

```
integer li_Index
li_Index = lb_actions.SelectedIndex( )
lb_actions.DeleteItem(li_Index)
```

The following statement deletes the item "Personal" from the ListBox lb_purpose:

```
lb_purpose.DeleteItem( &
   lb_purpose.FindItem( "Personal", 1))
```

See also

AddItem
FindItem
InsertItem
SelectItem

DeleteRow

Description

Deletes a row from a DataWindow control or child DataWindow.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**DeleteRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to delete a row.
row	A long identifying the row you want to delete. To delete the current row, specify 0 for *row*.

Return value

Integer. Returns *1* if the row is successfully deleted and *-1* if an error occurs.

Usage

DeleteRow deletes the row from the DataWindow's primary buffer.

If the DataWindow control is not updatable, all storage associated with the row is cleared. If the DataWindow is updatable, DeleteRow moves the row to the DataWindow's delete buffer; PowerBuilder uses the values in the delete buffer to build the SQL DELETE statement.

The row is not deleted from the database table until the application calls the Update function. After the Update function has updated the database and the update flags are reset, then the storage associated with the row is cleared.

Examples

This statement deletes the current row from dw_employee:

```
dw_employee.DeleteRow(0)
```

These statements delete row 5 from dw_employee and then update the database with the change:

```
dw_employee.DeleteRow(5)
dw_employee.Update( )
```

See also

DeletedCount
ResetUpdate
InsertRow
Retrieve
Update

DeleteSeries

Description

Deletes a series and its data values from a graph.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**DeleteSeries** (*seriesname*)

Parameter	Description
controlname	The graph in which you want to delete a series
seriesname	A string whose value is the name of the series you want to delete from *controlname*

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The series in a graph are numbered consecutively, in the order they were added to the graph. When a series is deleted, the remaining series are renumbered.

Example

This script for the SelectionChanged event of a DropDownListBox assumes that the listbox lists the series in the graph gr_data. When the user chooses an item, DeleteSeries deletes the series from the graph and DeleteItem deletes the name from the listbox:

```
string ls_name
ls_name = This.Text
gr_data.DeleteSeries(ls_name)
This.DeleteItem(This.FindItem(ls_name, 0))
```

See also

AddSeries
DeleteCategory
DeleteData
FindSeries

Describe

Description

Reports the values of attributes of a DataWindow object and objects within the DataWindow object. Each column and graphic object in the DataWindow has a set of attributes, which are listed in Appendix A. You specify one or more attributes as a string and Describe returns the values of the attributes.

Describe can also evaluate expressions involving values of a particular row and column. When you include Describe's Evaluate function in the attribute list, the value of the evaluated expression is included in the reported information.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Describe** (*attributelist*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want information about the structure.
attributelist	A string whose value is a blank-separated list of attributes or Evaluate functions. For a list of valid attributes, see Appendix A, "DataWindow Object Attributes."

Return value

String. Returns a string that includes a value for each attribute or Evaluate function. A newline character (~n) separates the value of each item in *attributelist*.

If the attribute list contains an invalid item, Describe returns an exclamation point (!) for that item and ignores the rest of the attribute list. Describe returns a question mark (?) if there is no value for an attribute.

When the value of an attribute contains a question mark (?), the value is returned in quotes, so that you can distinguish between it and an attribute with no value.

Usage

Use Describe to understand the structure of a DataWindow. For example, you can find out which bands the DataWindow uses and the data types of the columns. You can also use Describe to find out the current value of an attribute and use that value to make further modifications.

Describe is often used to obtain the DataWindow's SELECT statement in order to modify it (for example, by adding a WHERE clause).

When you can obtain the DataWindow's SQL statement
When you use the Select painter to graphically create a SELECT statement, PowerBuilder saves its own SELECT statement (called a PBSELECT statement) with the DataWindow definition, not a SQL SELECT statement. When you call Describe with the attribute Table.Select, it returns a SQL SELECT statement *only if* you are connected to the database. If you are not connected to the database, Describe returns a PBSELECT statement.

Attribute syntax

The syntax for an attribute in the attribute list is:

objectname.attribute

See Appendix A for the types of objects in a DataWindow and their attributes. The appendix includes examples for individual attributes.

Attributes whose value is a list

When an attribute returns a list, the tab character (~t) separates the values in the list. For example, the Bands attribute reports all the bands in use in the DataWindow:

header~tdetail~tsummary~tfooter~theader.1~ttrailer.1

If the first character in an attribute's returned value list is a quotation mark, it means the whole list is quoted and any quotation marks within the list are single quotation marks. For example, the following is a single attribute value:

" Student~T'Andrew'or'~NAndy' "

Quoted attribute values
Describe returns an attribute's value enclosed in quotes when the text would otherwise be ambiguous. For example, if the attribute's value includes a question mark, then the text is returned in quotes. A question mark without quotes means that the attribute has no value.

Column name or number

When the object is a column, you can specify the column name or a pound sign (#) followed by the column number. For example, if salary is column 5, then "salary.coltype" is equivalent to "#5.coltype".

Object names

The DataWindow painter automatically gives names to columns and column labels. Other objects that you add to the DataWindow object are named with a cryptic string of numbers unless you give them names. (Describe will report the cryptic names, but you can't see them in the painter.) To easily describe and modify attributes of an object, give the object a name.

Evaluating an expression

Describe's Evaluate function allows you to evaluate DataWindow painter expressions within a script using data in the DataWindow. Evaluate has the following syntax, which you specify for *attributelist*:

Evaluate('*expression*', *rownumber*)

Expression is the expression you want to evaluate and *rownumber* is the number of the row for which you want to evaluate the expression. The expression usually includes DataWindow painter functions. For example, in the following statement, Describe reports either 255 or 0 depending on the value of the salary column in row 3:

```
ls_ret = dw_1.Describe(  &
   "Evaluate('If(salary > 100000, 255, 0)', 3)")
```

You can call DataWindow control functions in a script to get data from the DataWindow, but some painter functions, such as LookUpDisplay, cannot be called in a script. Using Evaluate is the only way to call them. (See the example "Evaluating the display value of a DropDownDataWindow.")

Sample attribute values

To illustrate the types of values that Describe reports, consider a DataWindow called dw_emp with one group level. Its columns are named emp and empname, and its headers are named emp_h and empname_h. The following table shows several attributes and the returned value. In the first example below, a sample command shows how you might specify these attributes for Describe and what it reports.

Attribute	Reported Value
datawindow.Bands	header~tdetail~tsummary~tfooter~theader.1~ttrailer.1
datawindow.Objects	emp~tempname~temp_h~tempname_h~t
	If the object does not have a name, PowerBuilder assigns the object a numeric name, such as obj_1234567.
emp.Type	column
empname.Type	column
empname_h.Type	text
emp.Coltype	char(20)
state	! (! indicates an invalid item — there is no column named state.)
empname_h.Visible	?

Examples

The following example calls Describe with the list of attributes shown in the previous table. The reported values, formatted with tabs and newlines, follows. Note that because state is not an object in the DataWindow, state.type returns an exclamation point (!).

```
string ls_request, ls_report

ls_request = "DataWindow.Bands DataWindow.Objects "&
   +"empname_h.Text "  &
   +"empname_h.Type emp.Type emp.Coltype "  &
   +"state.Type empname.Type empname_h.Visible"

ls_report = dw_1.Describe(ls_request)
```

Describe sets the value of ls_report to the following string:

```
header~tdetail~tsummary~tfooter~theader.1~ttrailer.1
~N emp~tempname~temp_h~tempname_h~N
"Employee~R~NName"cd~N text~N column~Nchar(20)~N!
```

These statements check the data type of the column named salary before using GetItemNumber to obtain the salary value:

```
string ls_data_type
integer li_rate

ls_data_type = dw_1.Describe("salary.ColType")
IF ls_data_type = "number" THEN
   li_rate = dw_1.GetItemNumber(5, "salary")
ELSE
   . . .  // Some processing
END IF
```

Column name or number

The following statement finds out the column type of the current column, using the column name:

```
s = This.Describe(This.GetColumnName()+ ".ColType")
```

For comparison, the following statement finds out the same thing, using the current column's number:

```
s = This.Describe("#" + String(This.GetColumn( ))  &
   + ".ColType")
```

Scrolling and the current row

This example, as part of the DataWindow control's ScrollVertical event, makes the first visible row the current row as the user scrolls through the DataWindow:

```
s = This.Describe("DataWindow.FirstRowOnPage")
IF IsNumber(s) THEN This.SetRow(Integer(s))
```

Evaluating the display value of a DropDown-DataWindow

This example uses Describe's Evaluate function to find out the display value in a DropDownDataWindow column called state_code. Note that you must execute the code *after* the ItemChanged event, so that the value the user selected has become the item value in the buffer. This code is the script of a custom user event called getdisplayvalue:

```
string rownumber, displayvalue

rownumber = String(dw_1.GetRow())
displayvalue = dw_1.Describe(  &
   "Evaluate('LookUpDisplay(state_code) ', "  &
   + rownumber + ")")
```

This code, as part of the ItemChanged event's script, posts the getdisplayvalue event:

```
dw_1.PostEvent("getdisplayvalue")
```

Assigning null values based on the column's data type

The following excerpt from the ItemError event script of a DataWindow control allows the user to blank out a column and move to the next column. For columns with data types other than string, the user cannot leave the value empty (which is an empty string and doesn't match the data type) without this code:

```
string s
s = This.Describe(This.GetColumnName( )  &
   + ".Coltype")

CHOOSE CASE s
   CASE "number"
      IF Trim(This.GetText( )) = "" THEN
         integer  null_num
         SetNull(null_num)
         This.SetItem(This.GetRow( ), &
            This.GetColumn( ), null_num)
         This.SetActionCode(3)
      END IF

   CASE "date"
      IF Trim(This.GetText()) = "" THEN
         date  null_date
         SetNull(null_date)
         This.SetItem(This.GetRow( ),&
            This.GetColumn( ),null_date)
         This.SetActionCode(3)
      END IF

   . . . // Additional cases for other data types

END CHOOSE
```

See also

Create
Modify

DirList

Description

Populates a listbox with a list of files. You can specify a path, a mask, and a file type to restrict the set of files displayed. If the window has an associated StaticText control, you can have DirList display the current drive and directory as well.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**DirList** (*filespec*, *filetype* {, *statictext* })

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox you want to populate.
filespec	A string whose value is the file pattern. This is usually a mask (for example, *.INI or *.TXT). If you include a path, it becomes the current drive and directory.
filetype	An integer representing one or more types of files you want to list in the ListBox. Types are: ♦ 0 — Read/write files ♦ 1 — Read-only files ♦ 2 — Hidden files ♦ 4 — System files ♦ 16 — Subdirectories ♦ 32 — Archive (modified) files ♦ 16384 — Drives ♦ 32768 — Exclude read-write files from the list To list several types, add the numbers associated with the types. For example, to list read-write files, subdirectories, and drives, use 0+16+16384 or 16400 for *filetype*.
statictext (optional)	The name of the StaticText in which you want to display the current drive and directory.

Return value

Boolean. Returns TRUE if the search path is valid so that the ListBox is populated or the list is empty. DirList returns FALSE if the ListBox could not be populated (for example, *filespec* was a file, not a directory, or specified an invalid path).

Usage

You can call DirList when the window opens to populate the list initially. You should also call DirList in the script for the SelectionChanged event to repopulate the ListBox based on the new selection. (See the example in DirSelect.)

> **Tip**
> Although DirList's features allow you to emulate the standard File Open and File Save windows, you can get the full functionality of these standard windows by calling GetFileOpenName and GetFileSaveName instead of DirList.

Examples

This statement populates the ListBox lb_emp with a list of read-write files with the file extension TXT in the search path C:\EMPLOYEE*.TXT:

```
lb_emp.DirList("C:\EMPLOYEE\*.TXT", 0)
```

This statement populates the ListBox lb_emp with a list of read-only files with the file extension DOC in the search path `C:\EMPLOYEE*.DOC and displays the path specification in the StaticText st_path:

```
lb_emp.DirList("C:\EMPLOYEE\*.DOC", 1, st_path)
```

These statements in the script for a window Open event initialize a ListBox to all files in the current directory that match *.TXT:

```
String s_filespec
s_filespec = "*.TXT"
lb_filelist.DirList(s_filespec, 16400, st_filepath)
```

See also

DirSelect

DirSelect

Description

When a listbox has been populated with the DirList function, DirSelect retrieves the current selection and stores it in a string variable.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**DirSelect** (*selection*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox from which you want to retrieve the current selection. The ListBox must have been populated using DirList, and the selection must be a drive letter, a file, or the name of a directory.
selection	A string variable in which the selected path name will be put.

Return value

Boolean. Returns TRUE if the current selection is a drive letter or a directory name, which can contain files and other directories, and FALSE if it is a file, indicating the user's final choice.

Usage

Use DirSelect in the SelectionChanged event to find out what the user chose. When the user's selection is a drive or directory, use the selection as a new directory specification for DirList.

Example

The following script for the SelectionChanged event for the ListBox lb_FileList call DirSelect to test whether the user's selection is a file. If not, the script joins the directory name with the file pattern, and calls DirList to populate the ListBox and display the current drive and directory in the StaticText st_FilePath. If the current selection is a file, other code processes the filename.

```
string ls_filename, ls_filespec = "*.TXT"

IF lb_FileList.DirSelect(ls_filename) THEN
   //If ls_filename is not a file,
   //append directory to ls_filespec.
   ls_filename = ls_filename + ls_filespec
   lb_filelist.DirList(ls_filename,  &
      16400, st_FilePath)
ELSE
   ...  //Process the file.
END IF
```

See also

DirList

Disable

Description

Disables an item on a menu. The menu item is dimmed (its color is changed to the user's disabled text color, usually gray), and the user cannot select it.

Applies to

MenuItems in a Menu object

Syntax

menuitem.**Disable** ( )

Parameter	Description
menuitem	The name of the MenuItem you want to deactivate (disable)

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Equivalent syntax

Setting the MenuItem's Enabled attribute is the same as calling Disable:

menuitem.Enabled = FALSE

This statement:

```
m_appl.m_edit.Enabled = FALSE
```

is equivalent to:

```
m_appl.m_edit.Disable( )
```

Example

This statement disables the m_edit MenuItem on the menu m_appl:

```
m_appl.m_edit.Disable( )
```

See also

Enable

DisconnectObject

Description

Releases any object that is connected to the specified OLEObject variable.

Applies to

OLEObject objects

Syntax

oleobject.**DisconnectObject** ()

Parameter	Description
oleobject	The name of an OLEObject variable which you want to disconnect from an OLE object. You cannot specify an OLEObject that is the Object attribute of an OLE 2.0 control.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Invalid call: the argument is the Object attribute of a control
- *-9* Other error

Usage

The OLEObject variable is used for OLE automation, in which the PowerBuilder application asks the server application to manipulate the OLE object programmatically.

For more information about OLE automation, see ConnectToObject.

Example

This example creates an OLEObject variable and connects it to a new Excel object. After some unspecified code, it then disconnects:

```
integer result
OLEObject myoleobject

myoleobject = CREATE OLEObject
result = myoleobject.ConnectToNewObject(  &
   "excel.application")
. . .
result = myoleobject.DisconnectObject( )
```

See also

ConnectToObject
ConnectToNewObject

DoScript

Description

Runs an AppleScript script. The AppleScript system extension must be installed on the Macintosh.

Platform information
DoScript only has an effect on Macintosh.

Syntax

DoScript (*script, result*)

Parameter	Description
script	A string specifying the script to be run. You can specify: ♦ The text of the script ♦ The name of a file containing the script text (the file type is 'TEXT') ♦ The name of a file containing a compiled script (the file type is 'osas')
result	The text returned by AppleScript. The result can be information produced by the script or error information if the script failed to complete.

Return value

Integer. Returns the result code returned from AppleScript. The result code is *0* if no error occurred. Returns *-1* if the AppleScript system extension is not installed..

Usage

You cannot use DoScript to run a script that has been saved as an application. Use the Run function instead.

Example

This example for the Clicked event of a Run Script button runs the script in the file My Script and displays the result in the MultiLineEdit mle_report. The return code is displayed in the SingleLineEdit sle_code:

```
integer rtn
string result

rtn = DoScript("My Script", result)
mle_report.Text = result
sle_code.Text = String(rtn)
```

This example runs the script that the user typed into the MultiLineEdit mle_script:

```
integer rtn
string result

rtn = DoScript(mle_script.Text, result)
```

Double

Description

Converts a string to a double or obtains a double value that is stored in a blob.

Syntax

Double (*stringorblob*)

Parameter	Description
stringorblob	The string whose value you want returned as a double or a blob in which the first value is the double value. The rest of the contents of the blob is ignored.

Return value

Double. Returns the contents of *stringorblob* as a double. If *stringorblob* is not a valid PowerScript number, Double returns *0*.

Usage

To distinguish between a string whose value is the number 0 and a string whose value is not a number, use the IsNumber function before calling the Double function.

Examples

This statement returns 24.372 as a double:

```
Double("24.372")
```

This statement returns the contents of the SingleLineEdit sle_distance as a double:

```
Double(sle_distance.Text)
```

After assigning blob data from the database to lb_blob, the following example obtains the double value stored at position 20 in the blob. (The length you specify for BlobMid must be at least as long as the value but can be longer):

```
double lb_num
lb_num = Double(BlobMid(lb_blob, 20, 40))
```

For an example of assigning and extracting values from a blob, see Real.

See also

Dec
Integer
Long
Real

DoVerb

Description

Requests the OLE server application to execute the specified verb for the OLE object in an OLE 2.0 control.

Applies to

OLE 2.0 controls

Syntax

ole2control.**DoVerb** (*verb*)

Parameter	Description
ole2control	The name of the OLE 2.0 control containing the object for which you want to execute a verb.
verb	An integer identifying a verb known to the OLE server application. Verbs are operations that the server can perform on the OLE object. Check the documentation for the server's OLE implementation to find out what verbs it supports.

Return value

Integer. Returns *0* if it succeeds and one of the following values if an error occurs:

- *-1* Control is empty
- *-2* Invalid verb for object
- *-3* Verb not implemented by object
- *-4* No verbs supported by object
- *-5* Object can't execute verb now
- *-9* Other error

Example

This example executes verb 7 for the object in the OLE 2.0 control ole_1:

```
integer result
result = ole_1.DoVerb(7)
```

See also

Activate
OLEActivate
SelectObject

Drag

Description

Starts or ends the dragging of a control.

Applies to

All controls except drawing objects (lines, ovals, rectangles, and rounded rectangles)

Syntax

control.**Drag** (*dragmode*)

Parameter	Description
control	The name of the control you want to drag or stop dragging (the source object).
dragmode	A value of the DragMode data type indicating the action you want to take on control: ♦ Begin! — Put *control* in Drag mode ♦ Cancel! — Stop dragging *control* but do not cause a DragDrop event ♦ End! — Stop dragging *control* and if *control* is over a target object, cause a DragDrop event

Return value

Integer. For all controls except OLE 2.0 controls, returns *1* if it succeeds and *-1* if you try to nest drag events or try to cancel the drag when *control* is not in Drag mode. The return value is usually not used.

For OLE 2.0 controls, returns the following values:

- *2* Object was Moved
- *1* Drag was canceled
- *0* Drag succeeded
- *-1* Control is empty
- *-9* Unspecified error

Usage

To see the list of draggable controls, choose the Browse Class Hierarchy command in the Library painter. All the objects in the hierarchy below dragobject are draggable.

If you set the control's DragAuto attribute to TRUE, PowerBuilder automatically puts the control in Drag mode when the user clicks it. The user must hold the mouse button down to drag.

When you use Drag to manually put a control in Drag mode, the user can drag the control by moving the mouse without holding down the mouse button. Clicking the left mouse button ends the drag. CANCEL! and END! are required *only* if you want to end the drag without requiring the user to click the left mouse button.

To make something happen when the user drags a control onto a target object, write scripts for one or more of the target's drag events (DragDrop, DragEnter, DragLeave, and DragWithin).

Example

This statement puts sle_emp into Drag mode:

```
sle_emp.Drag(Begin!)
```

See also

DraggedObject

DraggedObject

Description

Returns a reference to the control that triggered a drag event.

Syntax

DraggedObject ( )

Return value

DragObject, a special data type that includes all draggable controls (all the controls but no drawing objects). Returns a reference to the control that is currently being dragged.

> **Note**
> If no control is being dragged, an execution error message is displayed.

Usage

Call DraggedObject in a drag event for the target object. The drag events are DragDrop, DragEnter, DragLeave, and DragWithin.

Use TypeOf to obtain the data type of the control. To access the attributes of the control, you can assign the DragObject reference to a variable of that control's data type (see the example).

Example

These statements set which_control equal to the data type of the control that is currently being dragged, and then set ls_text_value to the text attribute of the dragged control:

```
SingleLineEdit sle_which
CommandButton cb_which
string ls_text_value
DragObject which_control

which_control = DraggedObject( )

CHOOSE CASE TypeOf(which_control)

CASE CommandButton!
   cb_which = which_control
   ls_text_value = cb_which.Text

CASE SingleLineEdit!
   sle_which = which_control
   ls_text_value = sle_which.Text
END CHOOSE
```

See also

Drag
TypeOf

Draw

Description

Draws a picture control at a specified location in the current window.

Applies to

Picture controls

Syntax

picture.**Draw** (*xlocation*, *ylocation*)

Parameter	Description
picture	The name of the picture control you want to draw in the current window
xlocation	The x coordinate of the location (in PowerBuilder units) at which you want to draw the picture
ylocation	The y coordinate of the location (in PowerBuilder units) at which you want to draw the picture

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

Using the Draw function is faster and produces less flicker than successively changing the X attribute of a picture. This is because the Draw function draws directly on the window rather than having to recreate a small window with the picture in it for each change. Therefore, use Draw to draw pictures in animation.

To create animation, you can place *picture* outside the visible portion of the window and then use the Draw function to draw it at different locations in the window. However, the image remains at all the positions that you draw it. If you change the position by small increments, each new drawing of the picture covers up most of the previous image.

Using Draw does not change the position of the picture control—it just displays the control's image at the specified location. Use the Move function to actually change the position of the control.

Examples

This statement draws the bitmap p_Train at the location specified by the X and Y coordinates 100 and 200:

```
p_Train.Draw(100, 200)
```

These statements draw the bitmap p_ Train in a number of different locations so it appears to move from left to right across the window:

```
integer horizontal
FOR horizontal = 1 to 2000 step 8
   p_Train.Draw(horizontal, 100)
NEXT
```

See also

Move

Enable

Description

Enables an item on a menu so a user can select it.

Applies to

MenuItems in Menu objects

Syntax

menuitem.**Enable ( )**

Parameter	Description
menuitem	The name of the MenuItem you want to enable

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Enabling a menu item changes its color to the active color, not the dimmed, or disabled, color.

Calling Enable sets the item's Enabled attribute to TRUE.

Equivalent syntax

Setting the MenuItem's Enabled attribute is the same as calling Enable:

menuitem.Enabled = TRUE

This statement:

```
menu_appl.m_delete.Enabled = TRUE
```

is equivalent to:

```
menu_appl.m_delete.Enable( )
```

Example

This statement enables the m_delete MenuItem on the menu m_appl:

```
m_appl.m_delete.Enable( )
```

See also

Disable

EventParmDouble

Description

Retrieves a numeric value returned by a VBX standard or custom event and stores it in a PowerBuilder variable.

Applies to

VBX user objects

Syntax

vbxuserobject.**EventParmDouble** (*parameter*, *parmvariable*)

Parameter	Description
vbxuserobject	The name of the VBX user object for which you want the parameter of the VBX event
parameter	The number of the parameter of the VBX event for which you want the value
parmvariable	A double variable in which you want to store the parameter value

Return value

Integer. Returns *1* if it succeeds and *_1* if an error occurs.

Usage

Call this function when you want to use a numeric parameter returned by a VBX standard or custom event. You can use this function only for VBX user objects.

For information about custom events, see the documentation that was provided with the VBX control.

Example

This statement retrieves a numeric event parameter from a VBX control and stores in button_nbr:

```
Double button_nbr
vbx_control.EventParmDouble(1, button_nbr)
```

See also the VBX example in the User Object samples that come with PowerBuilder.

See also

EventParmString

EventParmString

Description

Retrieves a string value returned by a VBX standard or custom event and stores it in a PowerBuilder variable.

Applies to

VBX user objects

Syntax

vbxuserobject.**EventParmString** (*parameter*, *parmvariable*)

Parameter	Description
vbxuserobject	The name of the VBX user object for which you want the parameter of the VBX event
parameter	The number of the parameter of the VBX event for which you want the value
parmvariable	A string variable in which you want to store the parameter value

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Call this function when you want to use a string parameter returned by a VBX standard or custom event. You can use this function only for VBX user objects.

For information about custom events, see the documentation that was provided with the VBX control.

Example

This example retrieves a string event parameter from a VBX control and stores it in action:

```
string action
vbx_control.EventParmString(1, action)
```

See also the VBX example in the User Object samples that come with PowerBuilder.

See also

EventParmDouble

ExecRemote

Description

Asks a DDE server application to execute the specified command. There are two syntaxes:

- When you want to send a single command to a DDE server application, called a cold link, use Syntax 1.

- When you have already called OpenChannel and established a warm link with a DDE server application, use Syntax 2.

Platform information
This and other DDE functions have no effect on the Macintosh.

Syntax 1

ExecRemote (*command, applname, topicname*)

Parameter	Description
command	A string whose value is the command you want a DDE server application to execute. Check the documentation for the server application to determine the correct command format.
applname	A string whose value is the DDE name of the server application.
topicname	A string identifying the data or the instance of the DDE application you want to use with the command. In Microsoft Excel, for example, the topic name could be system or the name of an open spreadsheet.

Return value 1

Integer. Returns *1* if it succeeds. If it fails, it returns a negative integer. Possible values are:

-1 Link was not started
-2 Request denied
-3 Could not terminate server

Syntax 2

ExecRemote (*command, handle* {, *windowhandle*})

Parameter	Description
command	A string whose value is the command you want a DDE server application to execute. The format of the command depends on the DDE application you want to execute the command.
handle	A long that identifies the channel to the DDE server application. The OpenChannel function returns *handle* when you call it to open a DDE channel.
windowhandle (optional)	The handle to the window that you want to act as the DDE client. Specify this parameter to control which window is acting as the DDE client when you have more than one open window. If you don't specify *windowhandle*, the active window acts as the DDE client.

Return value 2

Integer. Returns *1* if it succeeds. If an error occurs, ExecRemote returns a negative integer. Possible values are:

-1 Link was not started
-2 Request denied
-9 handle is NULL

Usage

The DDE server application must already be running when you call a DDE function. Use the Run function to start the application, if necessary.

The ExecRemote function allows you start a cold link or warm link between the PowerBuilder client application and the DDE server application.

A cold link is a single DDE command and is not associated with a DDE channel. Each time you call ExecRemote without opening a channel (Syntax 1), Windows polls all running applications to find the one that will acknowledge the request. This is also true for the related functions GetRemote or SetRemote.

A warm link is associated with a DDE channel. You establish a channel for the DDE conversation with OpenChannel before sending commands with Syntax 2 of ExecRemote. A warm link is useful when you need to send several commands to the DDE server application. Because the channel is open, ExecRemote does not need to have Windows poll all running applications again. After you have called ExecRemote or the related functions GetRemote or SetRemote, and finished the work with the DDE server, call CloseChannel to end the DDE conversation.

A DDE hot link, which enables automatic updating of data in the PowerBuilder client application, involves other functions. See the StartHotLink function for more information.

Examples

This statement asks Microsoft Excel to save the active spreadsheet as file REGION.XLS. A channel is not open, so the function arguments specify the application and topic (the name of the spreadsheet):

```
ExecRemote("[Save( )]", "Excel", "REGION.XLS")
```

Syntax 2

This excerpt from a script asks the DDE channel to Microsoft Excel to save the active spreadsheet as file REGION.XLS. The OpenChannel function names the server application and the topic, so ExecRemote only needs to specify the channel handle. The script is associated with a button on a window, whose handle is specified as the last argument of OpenChannel:

```
long handle

handle = OpenChannel("Excel", "REGION.XLS", &
      Handle(Parent))

. . .  // Some processing

ExecRemote("[Save]", handle)
CloseChannel(handle, Handle(Parent))
```

See also

CloseChannel
GetRemote
OpenChannel
SetRemote

Exp

Description

Raises *e* to the specified power.

Syntax

Exp (*n*)

Parameter	Description
n	The power to which you want to raise e (2.71828)

Return value

Double. Returns *e* raised to the power *n*.

Examples

This statement returns 7.38905609893065:

```
Exp(2)
```

These statements convert a natural logarithm (base e) back to a regular number. When executed, Exp sets value to 200:

```
double value, x = log(200)
value = Exp(x)
```

See also

Log
LogTen
Exp in Chapter 2, "DataWindow Painter Functions"

Fact

Description

Determines the factorial of a number.

Syntax

Fact (*n*)

Parameter	Description
n	The number for which you want the factorial

Return value

Double. Returns the factorial of *n*.

Examples

This statement returns 24 (that is, 1 * 2 * 3 * 4):

```
Fact(4)
```

Both these statements return 1:

```
Fact(1)
Fact(0)
```

See also

Fact in Chapter 2, "DataWindow Painter Functions"

FileClose

Description

Closes the file associated with the specified file number. The file number was assigned to the file with the FileOpen function.

Syntax

FileClose (*file#*)

Parameter	Description
file#	The integer assigned to the file you want to close. The FileOpen function returns the file number when it opens the file.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

These statements open and then close the file EMPLOYEE.DAT. The variable li_FileNum stores the number assigned to the file when FileOpen opens the file. FileClose uses that number to close the file.

```
integer li_FileNum
li_FileNum = FileOpen("EMPLOYEE.DAT")
. . .  // Some processing
FileClose(li_FileNum)
```

See also

FileLength
FileOpen
FileRead
FileWrite

FileDelete

Description

Deletes the named file.

Syntax

FileDelete (*filename*)

Parameter	Description
filename	A string whose value is the name of the file you want to delete

Return value

Boolean. Returns TRUE if it succeeds, FALSE if an error occurs.

Example

These statements delete the file the user selected in the Open File window:

```
integer ret, value
string docname, named

value = GetFileOpenName("Select File,"  &
   docname, named, "DOC",  &
   "Doc Files (*.DOC),*.DOC")

IF value = 1 THEN ret = MessageBox("Delete",  &
   "Delete file?", Question!, OKCancel!)

IF ret = 1 THEN FileDelete(docname)
```

See also

FileExists

FileExists

Description

Reports whether the specified file exists.

Syntax

FileExists (*filename*)

Parameter	Description
filename	A string whose value is the name of a file

Return value

Boolean. Returns TRUE if the file exists, FALSE if it does not exist.

Example

This example determines if the file the user selected in the Save File window exists and, if so, asks the user if it's OK to overwrite it:

```
string ls_docname, ls_named
integer li_ret
boolean lb_exist

GetFileSaveName("Select File," ls_docname,  &
   ls_named, "pbl",  &
   "Doc Files (*.DOC),*.DOC")

lb_exist = FileExists(ls_docname)
IF lb_exist THEN li_ret = MessageBox("Save",  &
   "OK to write over" + ls_docname,  &
   Question!, YesNo!)
```

See also

FileDelete

FileLength

Description

Reports the length of a file in bytes.

Syntax

FileLength (*filename*)

Parameter	Description
filename	A string whose value is the name of the file for which you want to know the length. If *filename* is not on the current application library search path, specify the fully qualified name.

Return value

Long. Returns the length in bytes of the file identified by *filename*. If the file does not exist, FileLength returns *-1*.

Usage

Call FileLength before or after you call FileOpen to check the length of a file before you call FileRead. The FileRead function can read a maximum of 32765 characters at a time.

> **File security**
> If any security is set for the file (for example, if you are sharing the file on a network), you must call FileLength before FileOpen or after FileClose. Otherwise, you will get a sharing violation.

Examples

This statement returns the length of the file EMPLOYEE.DAT in the current directory:

```
FileLength("EMPLOYEE.DAT")
```

These statements determine the length of the EMP.TXT file in the EAST directory and open the file:

```
long LengthA
integer li_FileNum

LengthA = FileLength("C:\EAST\EMP.TXT")
li_FileNum = FileOpen("C:\EAST\EMP.TXT",  &
   StreamMode!, Read!, LockReadWrite!)
```

See also the examples for FileRead, which illustrate reading files of different lengths.

See also

FileClose
FileOpen
FileRead
FileWrite

FileOpen

Description

Opens the specified file for reading or writing and assigns it a unique integer file number. You use this integer to identify the file when you read, write, or close the file. The optional arguments *filemode*, *fileaccess*, *filelock*, and *writemode* determine the mode in which the file is opened.

Syntax

FileOpen (*filename*{, *filemode*{, *fileaccess*{, *filelock*{, *writemode* }}}})

Parameter	Description
filename	A string whose value is the name of the file you want to open. If *filename* is not on the operating system's search path, enter the fully qualified name.
filemode (optional)	A value of the FileMode enumerated type that specifies how the end of a FileRead or FileWrite is determined. Values are: ♦ LineMode! — (Default) Read or write the file a line at a time. ♦ StreamMode! — Read the file in 32K chunks. See Usage below.
fileaccess (optional)	A value of the FileAccess enumerated type that specifies whether the file is opened for reading or writing. Values are: ♦ Read! — (Default) Read only access. ♦ Write! — Write only access.
filelock (optional)	A value of the FileLock enumerated type specifying whether others have access to the opened file. Values are: ♦ LockReadWrite! — (Default) Only the user who opened the file has access. ♦ LockRead! — Only the user who opened the file can read it, everyone has write access. ♦ LockWrite! — Only the user who opened the file can write to it, everyone has read access. ♦ Shared! — All users have read and write access.
writemode (optional)	A value of the WriteMode enumerated data type. When *fileaccess* is Write!, specifies whether existing data in the file is overwritten. Values are: ♦ Append! — (Default) Write data to the end of the file. ♦ Replace! — Replace all existing data in the file. *Writemode* is ignored if the *fileaccess* argument is Read!

Return value

Integer. Returns the file number assigned to *filename* if it succeeds and *-1* if an error occurs.

Usage

When a file has been opened in line mode, each call to the FileRead function reads until it encounters a carriage return (CR), linefeed (LF), or end-of-file mark (EOF). Each call to FileWrite adds a CR and LF at the end of each string it writes.

When a file has been opened in stream mode, a call to FileRead reads the whole file (until it encounters an EOF) or 32,765 bytes, which ever is less. FileWrite writes a maximum of 32,765 bytes in a single call and does not add CR and LF characters.

File not found
If PowerBuilder doesn't find the file, it creates a new file, giving it the specified name.

Examples

This example uses the default arguments and opens the file EMPLOYEE.DAT for reading. The default settings are LineMode!, Read!, and LockReadWrite!. FileRead will read the file line-by-line and no other user will be able to access the file until it is closed:

```
integer li_FileNum
li_FileNum = FileOpen("EMPLOYEE.DAT")
```

The following example opens the file EMPLOYEE.DAT in the DEPT directory in stream mode (StreamMode!) for write only access (Write!). Existing data is overwritten (Replace!). No other users can write to the file (LockWrite!):

```
integer li_FileNum
li_FileNum = FileOpen("C:\DEPT\EMPLOYEE.DAT",  &
   StreamMode!, Write!, LockWrite!, Replace!)
```

See also

FileClose
FileLength
FileRead
FileWrite

FileRead

Description

Reads data from the file associated with the specified file number, which was assigned to the file with the FileOpen function.

Syntax

FileRead (*file#, variable*)

Parameter	Description
file#	The integer assigned to the file when it was opened.
variable	The name of the string or blob variable into which you want to read the data.

Return value

Integer. Returns the number of characters or bytes read. If an end-of-file mark (EOF) is encountered before any characters are read, FileRead returns *-100*. If the file is opened in LineMode and a CR or LF is encountered before any characters are read, FileRead returns *0*. If an error occurs, FileRead returns *-1*.

Usage

If the file is opened in Line mode, File Read reads a line of the file (that is, until it encounters a CR, LF, or EOF). It stores the contents of the line in the specified variable, skips the line-end characters, and positions the file pointer at the beginning of the next line.

If the file was opened in Stream mode, FileRead reads to the end of the file or the next 32,765 bytes, whichever is shorter. FileRead begins reading at the file pointer, which is positioned at the beginning of the file when the file is opened for reading. If the file is longer than 32,765 bytes, FileRead automatically positions the pointer after each read operation so that it is ready to read the next chunk of data.

FileRead can read a maximum of 32,765 characters at a time. Therefore, before calling the FileRead function, call the FileLength function to check the file length. If your system has file sharing or security restrictions, you may need to call FileLength before you call FileOpen.

Examples

This example reads the file EMP_DATA.TXT if it is short enough to be read with one call to FileRead:

```
integer li_FileNum
string ls_Emp_Input
long ll_FLength

ll_FLength = FileLength("C:\HR\EMP_DATA.TXT")
li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT",  &
   StreamMode!)
IF ll_FLength < 32767 THEN
   FileRead(li_FileNum, ls_Emp_Input)
END IF
```

The following example reads the file EMP_PIC1.BMP and stores the data in the blob Emp_Id_Pic. The number of bytes read is stored in li_bytes:

```
integer li_fnum, li_bytes
blob Emp_Id_Pic

li_fnum = FileOpen("C:\HR\EMP_PIC1.BMP",  &
   StreamMode!)
li_bytes = FileRead(li_fnum, Emp_Id_Pic)
```

The following example reads a file exceeding 32765 bytes. After the script has read the file into the blob tot_b, you can call the SetPicture or String function to make use of the data, depending on the contents of the file:

```
integer li_FileNum, loops, i
long flen, bytes_read, new_pos
blob b, tot_b

// Set a wait cursor
SetPointer(HourGlass!)

// Get the file length, and open the file
flen = FileLength(sle_filename.Text)
li_FileNum = FileOpen(sle_filename.Text,  &
   StreamMode!, Read!, LockRead!)

// Determine how many times to call FileRead
IF flen > 32765 THEN
   IF Mod(flen, 32765) = 0 THEN
      loops = flen/32765
   ELSE
      loops = (flen/32765) + 1
   END IF
ELSE
   loops = 1
END IF

// Read the file
new_pos = 1
FOR i = 1 to loops
   bytes_read = FileRead(li_FileNum, b)
   tot_b = tot_b + b
NEXT

FileClose(li_FileNum)
```

See also

FileClose
FileLength
FileOpen
FileSeek
FileWrite

FileSeek

Description

Moves the file pointer to the specified position in a file. The file pointer is the position in the file at which the next read or write begins.

Syntax

FileSeek (*file#, position, origin*)

Parameter	Description
file#	The integer assigned to the file when it was opened
position	A long whose value is the new position of the file pointer relative to the position specified in *origin*, in bytes
origin (optional)	The value of the SeekType enumerated data type specifying where you want to start the seek. Values are: ♦ FromBeginning! — (Default) At the beginning of the file ♦ FromCurrent! — At the current position ♦ FromEnd! — At the end of the file

Return value

Long. Returns the file position after the seek operation has been performed.

Usage

Use FileSeek to move within a binary file that you have opened in stream mode. FileSeek positions the file pointer so that the next FileRead or FileWrite occurs at that position within the file.

Examples

This example positions the file pointer 14 bytes from the end of the file:

```
integer li_FileNum

li_FileNum = FileOpen("emp_data")
FileSeek(li_FileNum, -14, FromEnd!)
```

This example moves the file pointer from its current position 14 bytes toward the end of the file. In this case, if no processing has occurred after FileOpen to affect the file pointer, specifying FromCurrent! is the same as specifying FromBeginning!:

```
integer li_FileNum

li_FileNum = FileOpen("emp_data")
FileSeek(li_FileNum, 14, FromCurrent!)
```

See also

FileRead
FileWrite

FileWrite

Description

Writes data to the file associated with the specified file number. The file number was assigned to the file with the FileOpen function.

Syntax

FileWrite (*file#, variable*)

Parameter	Description
file#	The integer assigned to the file when the file was opened.
variable	A string or blob whose value is the data you want to write to the file.

Return value

Integer. Returns the number of characters or bytes written if it succeeds and it returns *-1* if an error occurs.

Usage

FileWrite writes its data at the position identified by the file pointer. If the file was opened with the *writemode* argument set to Replace!, the file pointer is initially at the beginning of the file. After each call to FileWrite, the pointer is immediately after the last write. If the file was opened with the *writemode* argument set to Append!, the file pointer is initially at the end of the file and moves to the end of the file after each write.

FileWrite sets the file pointer to the character following the last character written. If the file was opened in line mode, FileWrite writes a carriage return (CR) and linefeed (LF) after the last character in *variable* and places the file pointer after the CR and LF.

> **Note**
> FileWrite can write only 32,766 bytes at a time, which includes the string terminator character. If the length of *variable* exceeds 32,765, FileWrite writes the first 32,765 characters and returns 32,765.

Examples

This script excerpt opens EMP_DATA.TXT and writes the string New Employees at the end of the file. The variable li_FileNum stores the number of the opened file:

```
integer li_FileNum

li_FileNum = FileOpen("C:\HR\EMP_DATA.TXT",  &
   LineMode!, Write!, LockWrite!, Append!)
FileWrite(li_FileNum, "New Employees")
```

The following example reads a blob from the database and writes it to a file. The SQL SELECT statement assigns the picture data to the blob Emp_Id_Pic. Then FileOpen opens a file for writing in stream mode and FileWrite writes the blob to the file. You could use the Len function to test whether the blob was too big for a single FileWrite call:

```
integer li_FileNum
blob emp_id_pic

SELECTBLOB salary_hist
   INTO  :emp_id_pic
   FROM Employee
   WHERE Employee.Emp_Num = 100
   USING Emp_tran;

li_FileNum = FileOpen(  &
   "C:\EMPLOYEE\EMP_PICS.BMP", &
   StreamMode!, Write!, Replace!)
FileWrite(li_FileNum, emp_id_pic)
```

See also

FileClose
FileLength
FileOpen
FileRead
FileSeek

Fill

Description

Builds a string of the specified length by repeating the specified characters until the result string is long enough.

Syntax

Fill (*chars*, *n*)

Parameter	Description
chars	A string whose value will be repeated to fill the return string
n	A long whose value is the length of the string you want returned

Return value

String. Returns a string *n* characters long filled with the characters in the argument *chars*. If the argument *chars* has more than *n* characters, the first *n* characters of *chars* are used to fill the return string. If the argument *chars* has fewer than *n* characters, the characters in *chars* are repeated until the return string has *n* characters.

Usage

Use Fill in printing routines to create a line or other special effect. For example, you can fill the amount line of a check with asterisks, or simulate a total line in a screen display by repeating hyphens below a column of figures.

Examples

This statement returns a string whose value is 35 stars:

```
Fill("*", 35)
```

This statement returns the string -+-+-+-:

```
Fill("-+", 7)
```

This statement returns 10 tildes (~):

```
Fill("~", 10)
```

See also

Space
Fill in Chapter 2, "DataWindow Painter Functions"

Filter

Description

Displays rows in a DataWindow that pass the current filter criteria. Rows that do not meet the filter criteria are moved to the filter buffer.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Filter ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that you want to filter

Return value

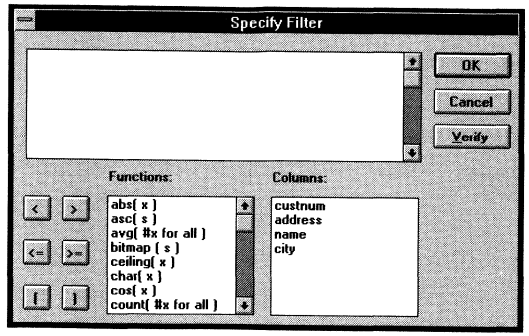
Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

Filter uses the current filter criteria for the DataWindow. To change the filter criteria, use the SetFilter function. The SetFilter function is equivalent to using the Filter command on the Rows menu of the DataWindow painter. If you do not call SetFilter to set the filter before you call Filter, Filter uses the filter specified in the DataWindow object definition.

When the Retrieve function retrieves data for the DataWindow, PowerBuilder applies the filter that was defined for the DataWindow object, if any. You only need to call Filter after you change the filter criteria with SetFilter or if the data has changed because of processing or user input.

To let users specify their own filter expression, pass a null string to the SetFilter function before you call Filter. Then when you call Filter, PowerBuilder displays the Specify Filter dialog with the filter expression blank.



Examples

This statement displays rows in dw_Employee based on its current filter criteria:

```
dw_Employee.Filter( )
```

The following example causes PowerBuilder to display the Specify Filter dialog because the filter expression has been set to null:

```
string null_str
SetNull(null_str)
dw_main.SetFilter(null_str)
dw_main.Filter( )
```

See also

FilteredCount
RowCount
SetFilter

FilteredCount

Description

Reports the number of rows that are not displayed in the DataWindow because of the current filter criteria.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**FilteredCount ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of rows that are filtered out of the display

Return value

Integer. Returns the number of rows in *datawindowname* that are not displayed because they do not meet the current filter criteria. Returns *0* if all rows are displayed and *-1* if an error occurs.

Usage

A DataWindow object can have a filter as part of its definition. After the DataWindow retrieves data, the filter is applied and rows that don't meet the filter criteria are moved to the filter buffer. You can change the filter criteria by calling the SetFilter function, and you can apply the new criteria with the Filter function.

Examples

These statements retrieve data in dw_Employee, display employees with area code 617, and then test to see if any other data was retrieved. If the filter criteria specifying the area code was part of the DataWindow definition, it would be applied automatically after calling Retrieve and you wouldn't need to call SetFilter and Filter:

```
dw_Employee.Retrieve()
dw_Employee.SetFilter("AreaCode=617")
dw_Employee.Filter()

// Did any rows get filtered out
IF dw_Employee.FilteredCount( ) > 0 THEN
   . . .  // Process rows not in area code 617
END IF
```

The following statements retrieve data in dw_Employee and display the number of employees whose names do not begin with B:

```
dw_Employee.Retrieve()
dw_Employee.SetFilter(  &
   "Left(emp_lname, 1)=~"B~"")
dw_Employee.Filter()

IF dw_Employee.FilteredCount( ) > 0 THEN
   MessageBox("Employee Count",  &
      String(dw_Employee.FilteredCount( )) +  &
      "Employee names do not begin with B.")
END IF
```

See also

Filter
ModifiedCount
RowCount
SetFilter

Find

Description

Finds the next row in a DataWindow in which data meets a specified condition.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Find** (*expression*, *start*, *end*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to search the detail band.
expression	A string whose value is a boolean expression that you want to use as the search criterion. The expression includes column names.
start	A long identifying the row location at which to begin the search.
end	A long identifying the row location at which to end the search. To search backward, make end less than start.

Return value

Long. Returns the number of the first row that meets the search criterion within the search range. Returns *0* if no rows are found and a negative number if an error occurs.

Usage

The search is case-sensitive. When you compare text to a value in a column, the case must match.

Examples

This statement searches for the first row in dw_status in which the value of the emp_salary column is greater than 100,000. The search begins in row 3 and continues until it reaches the last row in dw_status:

```
dw_status.Find( "emp_salary > 100000", 3,  &
   dw_status.RowCount( ))
```

To test values in more than one column, use boolean operators to join conditional expressions. The following statement searches for the employee named Smith whose salary exceeds 100,000:

```
dw_status.Find( &
   "emp_lname = 'Smith' and emp_salary > 100000",  &
   1, dw_status.RowCount( ) )
```

These statements search for the first row in dw_emp that matches the value that a user entered in the SingleLineEdit called Name. Note the single quotes embedded in the search expression around the name.

```
string ls_lname_emp
long ll_nbr, ll_foundrow

ll_nbr = dw_emp.RowCount( )

// Remove leading and trailing blanks.
ls_lname_emp = Trim(sle_Name.Text)

ll_foundrow = dw_emp.Find( &
   "emp_lname = '"+ ls_lname_emp +"'", 1, ll_nbr)
```

This script excerpt finds the first row that has a null value in emp_id. If no null is found, the script updates the DataWindow object. If a null is found, it displays a message:

```
IF dw_status.AcceptText( ) = 1 THEN
   IF dw_status.Find("IsNull(emp_id)",  &
         1, dw_status.RowCount( )) > 0 THEN
      MessageBox("Caution", "Cannot Update")
   ELSE
      dw_status.Update( )
   END IF
END IF
```

The following script attached to command button labeled Find Next searches for the next row that meets the specified criteria and scrolls to that row. Each time the button is clicked, the number of the found row is stored in the instance variable il_found. The next time the user clicks Find Next, the search continues from the following row. When the search reaches the end, a message tells the user that no row was found. The next search begins again at the first row.

Note that, although the search criteria is hard-coded here, a more realistic scenario would include a Find button that prompts the user for search criteria. You could store the criteria in an instance variable, which Find Next could use:

```
long ll_row

// Get the row number for the beginning of the
search
// from the instance variable, il_found
ll_row = il_found

// Search using predefined criteria
ll_row = dw_main.Find( &
   "item_id = 3 or item_desc = 'Nails'", &
   ll_row, dw_main.RowCount() )

IF ll_row > 0 THEN
   // Row found, scroll to it and make it current
   dw_main.ScrollToRow( ll_row )
ELSE
   // No row was found
   MessageBox("Not Found", "No row found.")
END IF

// Save the number of the next row for the start
// of the next search. If no row was found,
// ll_row is 0, making il_found 1, so that
// the next search begins again at the beginning
il_found = ll_row + 1
```

See also

FindGroupChange
FindRequired

FindCategory

Description

Obtains the number of a category in a graph when you know the category's label.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**FindCategory** ({ *graphcontrol*, } *categoryvalue*)

Parameter	Description
controlname	A string whose value is the name of the graph in which you want to find a specific category, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to find a specific category.
categoryvalue	A value that is the category for which you want the number. The value you specify must be the same data type as the data type of the category axis.

Return value

Integer. Returns the number of the category named in *categoryvalue* in the graph *controlname,* or if *controlname* is a DataWindow control, in *graphcontrol.* If an error occurs, FindCategory returns *-1.*

Usage

Most of the category manipulation functions require a category number, rather than a name. However, when you delete and insert categories, existing categories are renumbered to keep the numbering consecutive. Use FindCategory when you only know a category's label or when the numbering may have changed.

Examples

These statements obtain the number of a category in the graph gr_product_data. The category name is the text in the SingleLineEdit sle_category:

```
integer CategoryNbr
CategoryNbr =  &
   gr_product_data.FindCategory(sle_category.Text)
```

These statements obtain the number of the category named Qty in the graph gr_computers in the DataWindow control dw_equipment:

```
integer CategoryNbr
CategoryNbr = &
   dw_equipment.FindCategory("gr_computers", "Qty")
```

See also

AddCategory
DeleteData
DeleteSeries
FindSeries

FindGroupChange

Description

Searches for the next break for the specified group. A group break occurs when the value in the column for the group changes. FindGroupChange reports the row that begins the next section.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**FindGroupChange** (*row, level*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to search.
row	A long identifying the row at which you want to begin searching for the group break.
level	The number of the group for which you are searching. Groups are numbered in the order you defined them.

Return value

Long. Returns the number of the row whose group column has a new value, which begins a new group. Returns *0* if the value in the group column did not change and a negative number if an error occurs.

Usage

If the starting row begins a new section at the specified level, then that row is the one returned. To continue searching for subsequent breaks, increment the starting row so that the search resumes with the second row in the group.

Examples

This statement searches for the first break in group 2 in dw_regions. The search begins in row 5:

```
dw_regions.FindGroupChange(5, 2)
```

This code finds the number of the row at which a break occurs in group 1. It then checks whether the dept number is 121. The search begins at row 0:

```
boolean lb_found
long ll_breakrow

lb_found = FALSE
ll_breakrow = 0

DO WHILE NOT (lb_found)
   ll_breakrow = dw_1.FindGroupChange( &
         ll_breakrow, 1)
```

```
    // If no breaks are found, exit.
    IF ll_breakrow <= 0 THEN EXIT

    // Have we found the section for Dept 121?
    IF dw_1.GetItemNumber(ll_breakrow,  &
          "dept_id") = 121 THEN
       lb_found = TRUE
    END IF

    // Increment starting row to find next break
    ll_breakrow = ll_breakrow + 1
LOOP

IF lb_found = FALSE THEN
    MessageBox(  &
       "Not Found",  &
       "The Department was not found.")
ELSE
    . . .  // Processing for Dept 121
END IF
```

See also

Find
FindRequired

FindItem

Description

Finds the next item in a ListBox or DropDownListBox that begins with the specified search text.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**FindItem** (*text*, *index*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want to find an item
text	A string whose value is the starting text of the item you want to find
index	The number of the item at which you want to begin the search

Return value

Integer. Returns the index of the first matching item. To match, the item must start with the specified text; however, the text in the item can be longer than the specified text. If no match is found or if an error occurs, FindItem returns *-1*.

Usage

When FindItem finds the matching item, it returns the index of the item but does not select (highlight) the item. To find *and* select the item, use the SelectItem function.

Example

Assume the ListBox lb_actions contains the following list:

Index number	Item text
1	Open files
2	Close files
3	Copy files
4	Delete files

Then these statements start searching for Delete starting with item 2 (Close files). FindItem sets Index to 4:

```
integer Index
Index = lb_actions.FindItem("Delete", 2)
```

See also

AddItem
DeleteItem
InsertItem
SelectItem

FindRequired

Description

Reports the next row and column that is required and contains a NULL value. The function arguments that specify where to start searching also store the results of the search. You can speed up the search by specifying that FindRequired check only inserted and modified rows.

Applies to

DataWindow controls

Syntax

datawindowname.**FindRequired** (*dwbuffer, row, colnbr, colname, & updateonly*)

Parameter	Description
datawindowname	The name of the DataWindow control in which you want the find required columns that have NULL values.
dwbuffer	A value of the dwBuffer enumerated data type indicating the DataWindow buffer you want to search for required columns. Valid Buffers are: ♦ Primary! ♦ Filtered!
row	A long identifying the first row to be searched. Row also stores the number of the found row. FindRequired increments the row number automatically after it validates each row's columns. When it finds a row with a required column with a NULL value, the row number is stored in *row*. After FindRequired validates the last column in the last row, it sets *row* to 0.
colnbr	An integer specifying the first column to be searched. *Colnbr* also stores the number of the found column. After validating the last column, FindRequired sets *colnbr* to 1 and increments *row*. When it finds a required column without a NULL value, the column number is stored in *colnbr*.
colname	A string in which you want to store the name of the required column which contains a NULL value (the name of *colnbr*).
updateonly	A boolean indicating whether you want to validate all rows and columns or only rows that have been inserted or modified: ♦ TRUE — Validate only those that have changed. Setting *updateonly* to TRUE enhances performance in large DataWindows. ♦ FALSE — Validate all rows and columns.

Return value

Integer. Returns *1* if FindRequired successfully checked the rows and *-1* if an error occurs.

Usage

For FindRequired to report an empty required column, the column's value must actually be NULL, not an empty string.

To make a column required, set the Required attribute to TRUE in a script or check the Required checkbox in the DropDownListBox, Edit, or EditMask edit style window.

New rows have NULL values in their columns, unless the columns have default values. FindRequired will report empty required columns in new rows. When the user modifies a row and leaves a column empty, the new value will be an empty string, unless the column's edit style has the Empty String Is NULL checkbox checked. FindRequired will not report empty required columns in modified rows unless this attribute is set.

Examples

The following code makes a list of all the row numbers and column names in dw_1 in which required columns are missing values. The list is displayed in the MultiLineEdit mle_required:

```
long row = 1
integer colnbr
string colname

mle_required.Text = ""
DO WHILE row <> 0
   // If there's an error, exit
   IF dw_1.FindRequired(Primary!,  &
         row, colnbr,  &
         colname, FALSE) < 0 THEN EXIT

   // If a row was found, save the row and column
   IF row <> 0 THEN
      mle_required.Text = mle_required.Text  &
         + String(row) + "~t" &
         + colname + "~r~n"
   END IF

   // If no row was found, drop out of the loop
LOOP
```

This example is a function that ensures that no required column in a DataWindow control is empty (contains NULL). It takes one argument: the DataWindow control, which is declared:

```
DataWindow adw_control
```

The function returns -2 if the user's last entry can't be accepted or if FindRequired returns an error. It returns *-1* if an empty required column is found. It returns 1 if all required columns have data:

```
integer li_colnbr = 1
long ll_row = 1
string ls_colname, ls_textname

// Make sure the last entry is accepted
IF adw_control.AcceptText() = -1 THEN
   adw_control.SetFocus()
   RETURN -2
END IF

// Find the first empty row and column, if any
IF adw_control.FindRequired( Primary!, ll_row,  &
   li_colnbr, ls_colname, true ) < 0 THEN
   //If search fails due to error, then return
   RETURN -2
END IF

// Was any row found?
IF ll_row <> 0 THEN
   // Get the text of that column's label.
   ls_textname = ls_colname + "_t.Text"
   ls_colname = adw_control.Describe(ls_textname)

   // Tell the user which column to fill in.
   MessageBox("Required Value Missing",  &
      "Please enter a value for '"  &
      + ls_colname  &
      + "', row "  &
      + String(ll_row) + ".",
      StopSign! )

   // Make the problem column current.
   adw_control.SetColumn(li_colnbr)
   adw_control.ScrollToRow(ll_row)
   adw_control.SetFocus()
   RETURN -1
END IF

// Return success code if all required
// rows and columns have data
RETURN 1
```

See also

Find
FindGroupChange
MessageBox
ScrollToRow
SetColumn
SetTransObject

FindSeries

Description

Obtains the number of a series in a graph when you know the series' name.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**FindSeries** ({ *graphcontrol*, } *seriesname*)

Parameter	Description
controlname	The name of the graph containing the series for which you want the number, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control containing the series
seriesname	A string whose value is the name of the series for which you want the number

Return value

Integer. Returns the number of the series named in *seriesname* in the graph *controlname*, or if *controlname* is a DataWindow control, in *graphcontrol*. If an error occurs, FindSeries returns *-1*.

Usage

Most of the series manipulation functions require a series number, rather than a name. However, when you delete and insert series, existing series are renumbered so that the series are numbered consecutively. Use FindSeries when you only know a series' name or when the numbering may have changed.

Examples

These statements store the number of the series in the graph gr_product_data that was entered in the SingleLineEdit sle_series in SeriesNbr:

```
integer SeriesNbr
SeriesNbr = &
   gr_product_data.FindSeries(sle_series.Text)
```

These statements obtain the number of the series named PCs in the graph gr_computers in the DataWindow control dw_equipment and store it in SeriesNbr:

```
integer SeriesNbr
SeriesNbr = &
   dw_equipment.FindSeries("gr_computers", "PCs")
```

See also

AddSeries
DeleteSeries
FindCategory

GetActiveSheet

Description

Returns the currently active sheet in an MDI frame window.

Applies to

MDI frame windows

Syntax

mdiframewindow.**GetActiveSheet** ( )

Parameter	Description
mdiframewindow	The MDI frame window for which you want the active sheet

Return value

Window. Returns the sheet that is currently active in *mdiframewindow*. If no sheet is active, GetActiveSheet returns an invalid value.

Usage

Use the IsValid function to determine whether GetActiveSheet has returned a valid window value.

Example

These statements determine the active sheet in the MDI frame window w_frame and change the text of the MenuItem m_close on the menu m_file on the menu bar m_main. If no sheet is active, the text of the MenuItem is Close Window:

```
// Declare variable for active sheet
window activesheet
string mtext

activesheet = w_frame.GetActiveSheet( )
```

```
IF IsValid(activesheet) THEN
   // There is an active sheet, so get its title.
   // Change the text of the MenuItem to read
   // Close and the title of the active sheet.
   mtext = "Close " + activesheet.Title
   m_main.m_file.m_close.Text = mtext

ELSE
   // No sheet is active, MenuItem is Close Window
   m_main.m_file.m_close.Text = "Close Window"
END IF
```

See also IsValid

GetApplication

Description Gets the handle of the current Application object so you can get and set attributes of the application.

Syntax **GetApplication** ( )

Return value Application. Returns the handle of the current application object.

Usage The GetApplication function lets you write generic code for an application, making it reusable in other applications. You don't have to code the actual name of the application when you want to set application attributes.

Example To change whether Toolbar Tips are displayed, you can get the handle of the application object and set the ToolbarTips attribute:

```
application app
app = GetApplication( )
app.ToolbarTips = FALSE
```

The previous example could be coded more simply, as follows:

```
GetApplication( ).ToolbarTips = FALSE
```

GetBandAtPointer

Description

Reports the band in which the pointer is currently located, as well as the row number associated with the band. The bands are the headers, trailers, and detail areas of the DataWindow and correspond to the horizontal areas of the DataWindow painter.

Applies to

DataWindow controls

Syntax

datawindowname.**GetBandAtPointer** ( )

Parameter	Description
datawindowname	The name of the DataWindow control in which you want to obtain the band the pointer is located in

Return value

String. Returns a string that names the band in which the pointer is located, followed by a tab character (~t) and the number of the row associated with the band (see the table below). Returns the empty string ("") if an error occurs.

Usage

The following table lists the band names, where the pointer is when that band is reported, and the row that is associated with the band:

Band	Location of Pointer	Associated Row
detail	In the body of the DataWindow object.	The row at the pointer. If rows do not fill the body of the DataWindow object because of a group with a page break, then the first row of the next group. If the body isn't filled because there are no more rows, then the last row.
header	In the header of the DataWindow object.	The first row visible in the DataWindow body.
header.n	In the header of group level *n*.	The first row of the group.
trailer.n	In the trailer of group level *n*.	The last row of the group.
footer	In the footer of the DataWindow object.	The last row visible in the DataWindow body.

Band	Location of Pointer	Associated Row
summary	In the summary of the DataWindow object.	The last row before the summary.

You can parse the return value by searching for the tab character ("~t" or ASCII 09). For sample code that parses the return value, see the Pos function.

Example

These statements set the string named band to the location of the pointer in DataWindow dw_rpt:

```
String   band
band = dw_rpt.GetBandAtPointer( )
```

Some possible return values are:

Return value	Meaning
detail~t8	In row 8 of the detail band of dw_rpt
header~t10	In the header of dw_rpt; row 10 is the first visible row
header.2~t1	In the header of group level 2 for row 1
trailer.1~t5	In the trailer of group level 1 for row 5
footer~t111	In the footer of dw_rpt; the last visible row is 111
summary~t23	In the summary of dw_rpt; the last row is 23

See also

GetObjectAtPointer

GetBorderStyle

Description

Determines the border style of a column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetBorderStyle** (*column*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that contains the column.
column	The column for which you want to obtain the border style. *Column* can be a column number (integer) or a column name (string).

Return value

Border. Returns the border style of *column* in *datawindowname* as a value of the Border enumerated data type. Possible values are:

- Box!
- NoBorder!
- ShadowBox!
- Underline!

Returns NULL if it fails.

Example

These statements test the border of column 2 in dw_emp and, if there is no border, display a shadow box border:

```
border B2

B2 = dw_emp.GetBorderStyle(2)
IF B2 = NoBorder! THEN
   dw_emp.SetBorderStyle(2, ShadowBox!)
END IF
```

See also

SetBorderStyle

GetChild

Description

Provides a reference to a child DataWindow or to a report in a composite DataWindow, which you can use in DataWindow functions to manipulate that DataWindow or report.

Applies to

DataWindow controls

Syntax

datawindowname.**GetChild** (*name, dwchildvariable*)

Parameter	Description
datawindowname	The name of the DataWindow control that contains the child DataWindow or report
name	A string that names the column containing the child DataWindow or that names the report in the composite DataWindow
dwchildvariable	A variable of type DataWindowChild in which you want to store the reference to the child DataWindow

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The reference to the child DataWindow or report is stored in *dwchildvariable*.

Usage

A child DataWindow is a DropDownDataWindow in a DataWindow object.

A report is a DataWindow that it part of a composite DataWindow. A report is read-only. When you define the composite DataWindow in the DataWindow painter, make sure you give each report a name so that you can refer to it in the GetChild function.

Use GetChild when you need to explicitly retrieve data for a child DataWindow or report. Although PowerBuilder retrieves data for the child or report automatically when the main DataWindow is displayed, you need to explicitly retrieve data when there are retrieval arguments or when conditions change and you want to retrieve new rows.

When you insert a row or retrieve data in the main DataWindow, PowerBuilder automatically retrieves data for the child DataWindow. If the child DataWindow has retrieval arguments, PowerBuilder will display a dialog box asking the user for values for those arguments. To suppress the dialog box, you can explicitly retrieve data for the child before changing the main DataWindow. (See the example.)

Example

This example retrieves data for the child DataWindow associated with the column emp_state before retrieving data in the main DataWindow. The child DataWindow expects a region value as a retrieval argument. Because you populate the child DataWindow first, specifying a value for its retrieval argument, there is no need for PowerBuilder to display the retrieval argument dialog box:

```
DataWindowChild state_child
integer rtncode

rtncode = dw_1.GetChild('emp_state', state_child)
IF rtncode = -1 THEN MessageBox(  &
   "Error", "Not a DataWindowChild")

// Establish the connection if not already connected
CONNECT USING SQLCA;

// Set the transaction object for the child
state_child.SetTransObject(SQLCA)

// Populate the child with values for eastern states
state_child.Retrieve("East")

// Set transaction object for main DW and retrieve
dw_1.SetTransObject(SQLCA)
dw_1.Retrieve( )
```

In a composite DataWindow there are two reports: orders and current inventory. The orders report has a retrieval argument for selecting the order status. This report will display open orders. The composite DataWindow is displayed in a DataWindow control called dw_news and the reports are named open_orders and current_inv. The following code in the Open event of the window that contains dw_news provides a retrieval argument for open_orders:

```
DataWindowChild dwc_orders
dw_news.GetChild("open_orders", dwc_orders)
dwc_orders.SetTransObject(SQLCA)
dwc_orders.Retrieve("open")
```

See also

Handle
SetTransObject

GetClickedColumn

Description

Obtains the number of the column the user clicked or double-clicked in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetClickedColumn** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of the column the user clicked or double-clicked

Return value

Integer. Returns the number of the column that the user clicked or double-clicked in *datawindowname*. Returns *0* if the user did not click or double-click a column (for example, the user double-clicked outside the data area, in text or spaces between columns, or in the header, summary, or footer area).

Usage

Call GetClickedColumn in the Clicked or DoubleClicked event for a DataWindow control.

When the user clicks on the column, that column becomes the current column after the Clicked or DoubleClicked event is finished. During those events, GetColumn and GetClickedColumn can return different values.

If the user arrived at a column by another means, such as tabbing, GetClickedColumn will not tell you that column. Use GetColumn instead to find out the current column.

Example

These statements return the number of the column the user clicked or double-clicked in dw_employee:

```
integer li_ColNbr
li_ColNbr = dw_employee.GetClickedColumn( )
```

See also

GetClickedRow
GetColumn

GetClickedRow

Description

Obtains the number of the row the user clicked or double-clicked in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetClickedRow** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of the row the user clicked or double-clicked

Return value

Long. Returns the number of the row that the user clicked or double-clicked in *datawindowname*. Returns *0* if the user did not click or double-click a row (for example, the user double-clicked outside the data area, in text or spaces between rows, or in the header, summary, or footer area).

Usage

Call GetClickedRow in the Clicked or DoubleClicked event for a DataWindow control.

When the user clicks on the row, that row becomes the current row after the Clicked or DoubleClicked event is finished. During those events, GetRow and GetClickedRow can return different values.

If the user arrived at a row by another means, such as tabbing, GetClickedRow will not tell you that row. Use GetRow instead to find out the current row.

Example

These statements return the number of the row the user clicked or double-clicked in dw_Employee:

```
long ll_RowNbr
ll_RowNbr = dw_Employee.GetClickedRow( )
```

See also

GetClickedColumn
GetRow

GetColumn

Description

Obtains the number of the current column. The current column is the column that has the focus.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetColumn ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of the current column

Return value

Integer. Returns the number of the current column in *datawindowname*. Returns *0* if no column is current (because all the columns have a tab value of 0, making all of them uneditable), and *-1* if an error occurs.

Usage

GetColumn and GetClickedColumn, when called in the Clicked or DoubleClicked event, can return different values. The column the user clicked doesn't become current until after the event.

Use GetColumnName, instead of GetColumn, when you need the column's name. Use SetColumn to change the current column.

The current column

A column becomes the current column after the user tabs to it or clicks it or if a script calls the SetColumn function. A column cannot be current if it cannot be edited, that is, if it has a tab value of 0.

A DataWindow always has a current column, even when the control is not active, as long as there is at least one editable column.

Example

These statements return the number of the current column in dw_Employee:

```
Integer li_ColNbr
li_ColNbr = dw_Employee.GetColumn( )
```

See also

GetClickedColumn
GetColumnName

GetRow
SetColumn
SetRow

GetColumnName

Description

Obtains the name of the current column. The current column is the column that has the focus.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetColumnName ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the name of the current column

Return value

String. Returns the name of the current column in *datawindowname*. Returns the empty string ("") if no column is current or if an error occurs.

Usage

See GetColumn for information on the current column.

Example

These statements return the name of the current column in dw_Employee:

```
string ls_ColName
ls_ColName = dw_Employee.GetColumnName( )
```

See also

GetColumn
GetRow
SetColumn
SetRow

GetCommandDDE

Description

Obtains the command sent by the client application when your application is a DDE server.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

GetCommandDDE (*string*)

Parameter	Description
string	A string variable in which GetCommandDDE will store the command

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs (such as the function was called in the wrong context).

Usage

When a DDE client application sends a command to your application, the action triggers a RemoteExec event in the active window. In that event's script, you can call GetCommandDDEOrigin and GetCommandDDE to find out what application sent the command and what the command is. You decide how your application will respond to the command.

To enable DDE server mode, use the function StartServerDDE, in which you decide how your application will be known to other applications.

Example

This excerpt from a script for the RemoteExec event checks to see if the action requested by the DDE client is Open Next Sheet. If it is, the DDE server opens another instance of the sheet DataSheet. If the requested action is Shut Down, the DDE server shuts itself down. Otherwise, it lets the DDE client know the requested action was invalid.

The variables ii_sheetnum and i_DataSheet[] are instance variables for the window that responds to the DDE event:

```
integer ii_sheetnum
DataSheet i_DataSheet[ ]
```

The script below uses the local variable ls_Action to store the command sent by the client application:

```
string ls_Action

GetCommandDDE(ls_Action)
IF ls_Action = "Open Next Sheet" THEN
   ii_sheetnum = ii_sheetnum + 1
   OpenSheet(i_DataSheet[ii_sheetnum], w_frame_emp)
ELSEIF ls_Action = "Shut Down" THEN
   HALT CLOSE
ELSE
   RespondRemote(FALSE)
END IF
```

See also

GetCommandDDEOrigin
StartServerDDE
StopServerDDE

GetCommandDDEOrigin

Description

Obtains the name of the client application that sent a command to your application when your application is a DDE server.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

GetCommandDDEOrigin (*applstring*)

Parameter	Description
applstring	A string variable in which GetCommandDDEOrigin will store the name of the client application

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs (such as the function was called in the wrong context).

Usage

When a DDE client application sends a command to your application, the action triggers a RemoteExec event in the active window. In that event's script, you call GetCommandDDEOrigin and GetCommandDDE to find out what application sent the command and what the command is. You decide how your application will respond to the command.

You only need to use GetCommandDDEOrigin when more than one application may be acting as a DDE client.

Example

These statements illustrate one way to use GetCommandDDEOrigin in a multiclient situation:

```
string ls_WhichAppl
GetCommandDDEOrigin(ls_WhichAppl)
CHOOSE CASE ls_WhichAppl
   CASE "excel"
   ...  // Some processing specific to Excel
   CASE "winword"
   ...  // Processing for Word for Windows
   CASE "myapp"
   ...  // Processing specific to myapp
   CASE ELSE  //Not a valid client
      RespondRemote(FALSE)
END CHOOSE
```

See also

GetCommandDDE
StartServerDDE
StopServerDDE

GetData

Description

Obtains data from a control. Syntax 1, for graphs, obtains the value of a data point in a series. Syntax 2, for EditMask controls, obtains the unformatted data in the control.

Applies to

(Syntax 1) Graph controls in windows and user objects, and graphs in DataWindow controls

(Syntax 2) EditMask controls

Syntax 1

controlname.**GetData** ({ *graphcontrol*, } *seriesnumber*, & *datapoint* {, *datatype* })

Parameter	Description
controlname	The name of the graph from which you want data, or the name of the DataWindow control containing the graph.

Parameter	Description
graphcontrol (DataWindow control only)	A string whose value is the name of the graph from which you want the data when *controlname* is a DataWindow.
seriesnumber	The number that identifies the series from which you want data.
datapoint	The number of the data point for which you want the value.
datatype (scatter graph only)	A value of the grDataType enumerated data type specifying whether you want the x or y value of the data point in a scatter graph. Values are: ♦ xValue! — The x value of the data point ♦ yValue! — (Default) The y value of the data point

Return value 1

Double. Returns the value of the data in *datapoint* if it succeeds and *0* if an error occurs.

Syntax 2

editmaskname.**GetData** (*datavariable*)

Parameter	Description
editmaskname	The name of the EditMask control containing the data.
datavariable	A variable to which GetData will assign the unformatted data in the EditMask control. The data type of *datavariable* must match the data type of the EditMask control, which you select in the Window painter. Available data types are date, DateTime, decimal, double, string, or time.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

When you are working with EditMask controls (Syntax 2), you can find out the data type of an EditMask control by looking at its MaskDataType attribute, which holds a value of the MaskDataType enumerated data type.

Examples

These statements obtain the data value of data point 3 in the series named Costs in the graph gr_computers in the DataWindow control dw_equipment:

```
integer SeriesNbr
double data_value

// Get the number of the series.
SeriesNbr = &
   dw_equipment.FindSeries("gr_computers", "Costs")
data_value = dw_equipment.GetData(  &
   "gr_computers", SeriesNbr, 3)
```

These statements obtain the data value of the data point under the mouse pointer in the graph gr_prod_data and store it in data_value:

```
integer SeriesNbr, ItemNbr
double data_value
grObjectType MouseHit

MouseHit = &
   gr_prod_data.ObjectAtPointer(SeriesNbr, ItemNbr)
IF MouseHit = TypeSeries! THEN
   data_value = &
      gr_prod_data.GetData(SeriesNbr, ItemNbr)
END IF
```

These statements obtain the x value of the data point in the scatter graph gr_sales_yr and store it in data_value:

```
integer SeriesNbr, ItemNbr
double data_value

gr_product_data.ObjectAtPointer(SeriesNbr, ItemNbr)
data_value = &
   gr_sales_yr.GetData(SeriesNbr, ItemNbr, xValue!)
```

Syntax 2

This example gets data of data type date from the EditMask control em_date. Formatting characters for the date are ignored. The String function converts the date to a string so it can be assigned to the SingleLineEdit sle_date:

```
date d
em_date.GetData(d)
sle_date.Text = String(d, "mm-dd-yy")
```

This example gets string data from the EditMask control em_string and assigns the result to sle_string. Characters in the edit mask are ignored:

```
string s
em_string.GetData(s)
sle_string.Text = s
```

See also

DeleteData
FindSeries
InsertData
ObjectAtPointer

GetDataDDE

Description

Obtains data sent from another DDE application and stores it in the specified string variable. PowerBuilder can use GetDataDDE when acting as a DDE client or a DDE server application.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

GetDataDDE (*string*)

Parameter	Description
string	A string variable in which GetDataDDE will put the data received from a remote DDE application

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs (such as the function was called in the wrong context).

Usage

GetDataDDE is usually called in the window-level script for a RemoteSend event when your application is a DDE server or HotLinkAlarm event when your application is a DDE client.

Example

Assume that your PowerBuilder DDE client application has established a hot link with row 7, column 15 of an Excel spreadsheet, that the value in that row and column address has changed from red to green, which triggers the HotLinkAlarm event in your application. The script for the HotLinkAlarm event shown below calls GetDataDDE to store the new value in the variable Str20:

```
// In the script for a HotLinkAlarm event
string Str20
GetDataDDE(Str20)
```

See also

GetDataDDEOrigin
OpenChannel
StartServerDDE
StopServerDDE

GetDataDDEOrigin

Description

Determines the origin of data from a hot-linked DDE server application or a DDE client application, and if successful, stores the application's DDE identifiers in the specified strings. PowerBuilder can use GetDataDDEOrigin when it is acting as a DDE client or as a DDE server application.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

GetDataDDEOrigin (*applstring, topicstring, itemstring*)

Parameter	Description
applstring	A string variable in which GetDataDDEOrigin will store the name of the remote DDE application
topicstring	A string variable in which GetDataDDEOrigin will store the topic (for example, in Microsoft Excel, the topic could be REGION.XLS)
itemstring	A string variable in which GetDataDDEOrigin will store the item identification (for example, in Microsoft Excel, the item could be R1C2)

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs (such as the function was called in the wrong context).

Usage

Call GetDataDDEOrigin in the window-level script for a RemoteSend event or a HotLinkAlarm event.

When your application is a DDE server, call GetDataDDEOrigin in the script for the RemoteSend event. Use it to determine the source of the data when the data could come from more than one application.

When your application is a DDE client, call GetDataDDEOrigin in the script for the HotLinkAlarm event. Use it to identify the source of the data when hot links may exist for more than one topic within the server application or for more than one application.

Example

This example illustrates how to call GetDataDDEOrigin:

```
string WhichAppl, WhatTopic, WhatLoc
GetDataDDEOrigin(WhichAppl, WhatTopic, WhatLoc)
```

See also

GetDataDDE
OpenChannel
StartServerDDE
StopServerDDE

GetDataPieExplode

Description

Reports the percentage that a pie slice is exploded in a pie graph. An exploded slice is moved away from the center of the pie in order to draw attention to the data.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls.

Syntax

controlname.**GetDataPieExplode ({** *graphcontrol*, **}** *series*, & *datapoint*, *percentage* **)**

Parameter	Description
controlname	The name of the graph for which you want the percentage a pie slice is exploded, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the percentage a pie slice is exploded
seriesnumber	The number that identifies the series
datapoint	The number of the exploded data point (that is, the pie slice)
percentage	An integer variable in which you want to store the percentage that the pie slice is exploded

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

This example reports the percentage that a pie slice is exploded when the user clicks on that slice. The code checks whether the graph is a pie graph using the attribute GraphType. It then finds out whether the user clicked on a pie slice by checking the series and data point values set by ObjectAtPointer. The script is for the DoubleClicked event of a graph object:

```
integer series, datapoint
grObjectType  clickedtype
integer percentage

percentage = 50
IF (This.GraphType <> PieGraph! and  &
   This.GraphType <> Pie3D!) THEN RETURN
clickedtype = This.ObjectAtPointer(series,  &
   datapoint)

IF (series > 0 and datapoint > 0) THEN
   This.GetDataPieExplode(series, datapoint,  &
      percentage)
   MessageBox("Explosion Percentage",  &
   "Data point " + This.CategoryName(datapoint)  &
   + " in series " + This.SeriesName(series)  &
   + " is exploded " + String(percentage) + "%" )
END IF
```

See also

SetDataPieExplode

GetDataStyle

Description

Finds out the appearance of a data point in a graph. Each data point in a series can have individual appearance settings. There are different syntaxes, depending on what settings you want to check:

- To get the data point's colors, use Syntax 1.
- To get the line style and width, use Syntax 2.
- To get the fill pattern or symbol for the data point, use Syntax 3.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax 1

controlname.**GetDataStyle** ({ *graphcontrol*, } *seriesnumber*, *datapointnumber*, *colortype*, *colorvariable*)

Parameter	Description
controlname	The name of the graph for which you want the color of a data point, or the name of the DataWindow control containing the graph.
graphcontrol (Data Window control only)	When *controlname* is a DataWindow control, the name of the graph in the DataWindow control for which you want the color of a data point.
seriesnumber	The number of the series in which you want the color of a data point.
datapointnumber	The number of the data point for which you want the color.
colortype	A value of the grColorType enumerated data type specifying the aspect of the data point for which you want the color. Values are: ♦ Background! — The background color ♦ Foreground! — Text (fill color) ♦ LineColor! — The color of the line ♦ Shade! — The shaded area of three-dimensional graphics
colorvariable	A long variable in which you want to store the color.

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores a color value in *colorvariable*.

Syntax 2

controlname.**GetDataStyle** ({ *graphcontrol*, } & *seriesnumber*, *datapointnumber*, *linestyle*, *linewidth*)

Parameter	Description
controlname	The name of the graph for which you want the line style and width of a data point, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph (in the DataWindow control) for which you want the line style and width of a data point.

Parameter	Description
seriesnumber	The number of the series in which you want the line style and width of a data point.
datapointnumber	The number of the data point for which you want the line style and width.
linestyle	A variable of type LineStyle in which you want to store the line style.
linewidth	An integer variable in which you want to store the width of the line. The width is measured in pixels.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs. For the specified series and data point, stores its line style in *linestyle* and the line's width in *linewidth*.

Syntax 3

controlname.**GetDataStyle** ({ *graphcontrol*, } &
seriesnumber, *datapointnumber*, *enumvariable*)

Parameter	Description
controlname	The name of the graph for which you want the fill pattern or symbol type of a data point, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph (in the DataWindow control) for which you want the fill pattern or symbol type of a data point
seriesnumber	The number of the series in which you want the fill pattern or symbol type of a data point
datapointnumber	The number of the data point for which you want the fill pattern or symbol type.
enumvariable	The variable in which you want to store the data style. You can specify a FillPattern or grSymbolType variable. The data style information stored will depend on the variable type.

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores, according to the type of *enumvariable,* a value of that enumerated data type representing the fill pattern or symbol used for the specified data point.

Usage

GetDataStyle provides information about a single data point. The series to which the data point belongs has its own style settings. In general, the style values for the data point are the same as its series' settings. Use SetDataStyle to change the style values for individual data points. Use GetSeriesStyle and SetSeriesStyle to get and set style information for the series.

The graph stores style information for attributes that don't apply to the current graph type. For example, you can find out the fill pattern for a data point or a series in a two-dimensional line graph, but that fill pattern will not be visible.

See SetDataStyle for the enumerated data type values that GetDataStyle will store in *linestyle* and *enumvariable*.

Examples

This example gets the text (foreground) color used for data point 6 in the series named Salary in the graph gr_emp_data. It stores the color value in the variable color_nbr:

```
long color_nbr
integer SeriesNbr

// Get the number of the series
SeriesNbr = FindSeries("Salary")

// Get the color
gr_emp_data.GetDataStyle(SeriesNbr, 6,  &
   Foreground!, color_nbr)
```

This example gets the background color used for data point 6 in the series entered in the SingleLineEdit sle_series in the DataWindow graph gr_emp_data. It stores the color value in the variable color_nbr:

```
long color_nbr
integer SeriesNbr

// Get the number of the series
SeriesNbr =  &
   FindSeries("gr_emp_data", sle_series.Text)

// Get the color
dw_emp_data.GetDataStyle("gr_emp_data",  &
   SeriesNbr, 6, Background!, color_nbr)
```

Syntax 2

This example gets the line style and width of data point 10 in the series named Costs in the graph gr_product_data. It stores the information in the variables line_style and line_width:

```
integer SeriesNbr, line_width
LineStyle line_style

// Get the number of the series
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.GetDataStyle(SeriesNbr, 10,  &
   line_style, line_width)
```

This example gets the line style and width for data point 6 in the series entered in the SingleLineEdit sle_series in the graph gr_depts in the DataWindow control dw_employees. The information is stored in the variables line_style and line_width:

```
integer SeriesNbr, line_width
LineStyle line_style

// Get the number of the series
SeriesNbr = dw_employees.FindSeries(  &
   "gr_depts", sle_series.Text)

// Get the line style and width
dw_employees.GetDataStyle("gr_depts", SeriesNbr,  &
   6, line_style, line_width)
```

Syntax 3

This example gets the pattern used to fill data point 10 in the series named Costs in the graph gr_product_data. The information is stored in the variable data_pattern:

```
integer SeriesNbr
FillPattern data_pattern

// Get the number of the series
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.GetDataStyle(SeriesNbr, 10,  &
   data_pattern)
```

This example gets the pattern used to fill data point 6 in the series entered in the SingleLineEdit sle_series in the graph gr_depts in the DataWindow control dw_employees. The information is assigned to the variable data_pattern:

```
integer SeriesNbr
FillPattern data_pattern

// Get the number of the series
SeriesNbr = dw_employees.FindSeries("gr_depts",  &
   sle_series.Text)

// Get the pattern
dw_employees.GetDataStyle("gr_depts", SeriesNbr,  &
   6, data_pattern)
```

These statements store in the variable symbol_type the symbol of data point 10 in the series named Costs in the graph gr_product_data:

```
integer SeriesNbr
grSymbolType symbol_type

// Get the number of the series
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.GetDataStyle(SeriesNbr, 10,  &
   symbol_type)
```

These statements store the symbol for data point 6 in the series entered in the SingleLineEdit sle_series in the graph gr_depts in the DataWindow control dw_employees in the variable data_pattern:

```
integer SeriesNbr
grSymbolType symbol_type

// Get the number of the series
SeriesNbr = dw_employees.FindSeries("gr_depts",  &
   sle_series.Text)

// Get the symbol
dw_employees.GetDataStyle("gr_depts", SeriesNbr,  &
   6, symbol_type)
```

See also

FindSeries
GetSeriesStyle
SetDataStyle
SetSeriesStyle

GetDataValue

Description

Obtains the value of a data point in a series in a graph.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**GetDataValue** ({ *graphcontrol*, } *seriesnumber*, & *datapoint*, *datavariable* {, *xory* })

Parameter	Description
controlname	The name of the graph from which you want data, or the name of the DataWindow control containing the graph

Parameter	Description
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control from which you want the data
seriesnumber	The number that identifies the series from which you want data
datapoint	The number of the data point for which you want the value
datavariable	The name of a variable that will hold the data value. The variable's data type can be date, DateTime, double, string, or time. The variable must have the same data type as the values axis of the graph.
xory (scatter graph only)	A value of the grDataType enumerated data type specifying whether you want the x or y value of the data point in a scatter graph. Values are: ♦ xValue! — The x value of the data point ♦ yValue! — (Default) The y value of the data point

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Examples

These statements obtain the data value of data point 3 in the series named Costs in the graph gr_computers in the DataWindow control dw_equipment:

```
integer SeriesNbr, rtn
double data_value

// Get the number of the series.
SeriesNbr = dw_equipment.FindSeries(  &
   "gr_computers", "Costs")
rtn = dw_equipment.GetData(  &
   "gr_computers", SeriesNbr, 3, data_value)
```

These statements obtain the data value of the data point under the mouse pointer in the graph gr_prod_data and store it in data_value. If the user doesn't click on a data point, then ItemNbr is set to 0. The categories of the graph are time values:

```
integer SeriesNbr, ItemNbr, rtn
time data_value
grObjectType MouseHit

MouseHit = &
   gr_prod_data.ObjectAtPointer(SeriesNbr, ItemNbr)
IF ItemNbr > 0 THEN
   rtn = gr_prod_data.GetData(  &
      SeriesNbr, ItemNbr, data_value)
END IF
```

These statements obtain the x value of the data point in the scatter graph gr_sales_yr and store it in data_value. If the user doesn't click on a data point, then ItemNbr is set to 0. The data type of the category axis is Date:

```
integer SeriesNbr, ItemNbr, rtn
date data_value

gr_product_data.ObjectAtPointer(SeriesNbr, ItemNbr)
IF ItemNbr > 0 THEN
   rtn = gr_sales_yr.GetData(  &
      SeriesNbr, ItemNbr, data_value, xValue!)
END IF
```

See also

DeleteData
FindSeries
InsertData
ObjectAtPointer

GetDynamicDate

Description

Obtains data of type Date from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

> **Note**
> You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicDate** (*index*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.

Parameter	Description
index	An integer identifying the output parameter descriptor from which you want to get the data. Index must be less than or equal to the value in NumOutputs in DynamicDescriptionArea.

Return value

Date. Returns the Date data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea.* Returns *1900-01-01* if an error occurs.

Usage

After you fetch data using Format 4 dynamic SQL statements, the DynamicDescriptionArea, usually SQLDA, contains information about the data retrieved. The SQLDA attribute NumOutputs specifies the number of data descriptors returned. The attribute array OutParmType contains values of the ParmType enumerated data type specifying the data type of each value returned.

Use GetDynamicDate when the value of OutParmType is TypeDate! for the value in the array that you want to retrieve.

Examples

These statements set Today to the Date data in the second output parameter descriptor:

```
Date Today
Today = GetDynamicDate(SQLDA, 2)
```

If you have executed Format 4 dynamic SQL statements, data is stored in the DynamicDescriptionArea. This example finds out the data type of the stored data and uses a CHOOSE CASE statement to assign it to local variables.

If the SELECT statement is:

```
SELECT emp_start_date FROM employee;
```

then the code below at CASE Typedate! will be executed.

For each case, other processing could assign the value to a DataWindow so that the value would not be overwritten when another value has the same ParmType:

```
Date Datevar
Time Timevar
DateTime Datetimevar
Double Doublevar
String Stringvar
```

```
FOR n = 1 to SQLDA.NumOutputs
   CHOOSE CASE SQLDA.OutParmType[n]
      CASE TypeString!
         Stringvar = SQLDA.GetDynamicString(n)
         ...  // Other processing
      CASE TypeDecimal!, TypeDouble!,  &
            TypeInteger!, TypeLong!,  &
            TypeReal!, TypeBoolean!
         Doublevar = SQLDA.GetDynamicNumber(n)
         ...  // Other processing
      CASE TypeDate!
         Datevar = SQLDA.GetDynamicDate(n)
         ...  // Other processing
      CASE TypeDateTime!
         Datetimevar = SQLDA.GetDynamicDateTime(n)
         ...  // Other processing
      CASE TypeTime!
         Timevar = SQLDA.GetDynamicTime(n)
         ...  // Other processing
      CASE ELSE
         MessageBox("Dynamic SQL", &
               "Data type unknown.")
   END CHOOSE
NEXT
```

See also

Discussion of dynamic SQL in *PowerScript Language*
GetDynamicDateTime
GetDynamicNumber
GetDynamicString
GetDynamicTime
SetDynamicParm

GetDynamicDateTime

Description

Obtains data of type DateTime from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

Note
You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicDateTime** (*index*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.
index	An integer identifying the output parameter descriptor from which you want to get the data. *Index* must be less than or equal to the value in NumOutputs in DynamicDescriptionArea.

Return value

DateTime. Returns the DateTime data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea*. Returns *1900-01-01 00:00:00.000000* if an error occurs.

Usage

Use GetDynamicDateTime when the value of OutParmType is TypeDateTime! for the value that you want to retrieve from the array.

To test for the error value, you must use the DateTime function to construct the value to which you want to compare the returned value. PowerBuilder does not support DateTime literals.

Example

These statements set System to the DateTime data in the second output parameter descriptor:

```
DateTime SystemDateTime
SystemDateTime = SQLDA.GetDynamicDateTime(2)
IF SystemDateTime =  &
      DateTime(1900-01-01, 00:00:00) THEN
   ...  // Error handling
END IF
```

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

See also

Discussion of dynamic SQL in *PowerScript Language*
GetDynamicDate
GetDynamicNumber

GetDynamicString
GetDynamicTime
SetDynamicParm

GetDynamicNumber

Description

Obtains numeric data from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

> **Note**
> You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicNumber** (*index*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.
index	An integer identifying the output parameter descriptor from which you want to get the data. *Index* must be less than or equal to the value in NumOutputs in *DynamicDescriptionArea*.

Return value

A numeric data type (decimal, double, integer, long, or real). Returns the numeric data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea*. Returns *0* if an error occurs.

Usage

Use GetDynamicNumber when the value of OutParmType is TypeInteger!, TypeDecimal!, TypeDouble!, TypeLong!, TypeReal!, or TypeBoolean! for the value that you want to retrieve from the array.

Example

These statements set DeptId to the numeric data in the second output parameter descriptor:

```
Integer DeptId
DeptId = SQLDA.GetDynamicNumber(2)
```

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

See also

Discussion of dynamic SQL in *PowerScript Language*
GetDynamicDate
GetDynamicDateTime
GetDynamicString
GetDynamicTime
SetDynamicParm

GetDynamicString

Description

Obtains data of type String from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

> **Note**
> You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicString** (*index*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.
index	An integer identifying the output parameter descriptor from which you want to get the data. *Index* must be less than or equal to the value in NumOutputs in *DynamicDescriptionArea*.

Return value

String. Returns the string data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea*. Returns the empty string ("") if an error occurs.

Usage

Use GetDynamicString when the value of OutParmType is TypeString! for the value that you want to retrieve from the array.

Example

These statements set LName to the String data in the second output descriptor:

```
String LName
LName = SQLDA.GetDynamicString(2)
```

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

See also

Discussion of dynamic SQL in *PowerScript Language*
GetDynamicDate
GetDynamicDateTime
GetDynamicNumber
GetDynamicTime
SetDynamicParm

GetDynamicTime

Description

Obtains data of type Time from the DynamicDescriptionArea after you have executed a dynamic SQL statement.

> **Note**
> You can use this function *only* after executing Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**GetDynamicTime** (*index*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.
index	An integer identifying the output parameter descriptor from which you want to get the data. *Index* must be less than or equal to the value in NumOutputs in *DynamicDescriptionArea*.

Return value

Time. Returns the Time data in the output parameter descriptor identified by *index* in *DynamicDescriptionArea*. Returns *00:00:00.000000* if an error occurs.

Usage

Use GetDynamicTime when the value of OutParmType is TypeTime! for the value that you want to retrieve from the array.

Example

These statements set Start to the Time data in the first output parameter descriptor:

```
Time Start
Start = SQLDA.GetDynamicTime(1)
```

For an example of retrieving data from the DynamicDescriptionArea, see GetDynamicDate.

See also

Discussion of dynamic SQL in *PowerScript Language*
GetDynamicDate
GetDynamicDateTime
GetDynamicNumber
GetDynamicString
SetDynamicParm

GetEnvironment

Description

Gets information about the operating system, processor, and screen display of the system.

Syntax

GetEnvironment (*environmentinfo*)

Parameter	Description
environmentinfo	The name of the Environment object that will hold the information about the environment.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

In cross-platform development projects, you can call GetEnvironment in scripts and take actions based on the operating system (Windows 3.1, Macintosh, and so on). You can also find out the processor (Intel 386 or 486, 68000, and so on). The information also includes version numbers of the operating system and PowerBuilder.

You can call GetEnvironment to find out the number of colors supported by the system and the size of the screen. You can use the size information in a window's Open script to reset its X and Y attributes.

Example

The following script runs Excel, which is available on Windows and Macintosh, and uses the OSType attribute of the Environment object to determine how to specify the path:

```
string path
environment env
integer rtn

rtn = GetEnvironment(env)
IF rtn <> 1 THEN RETURN

CHOOSE CASE env.OSType
CASE Macintosh!
   path = "Macintosh HD:Excel Folder:Excel"
CASE Windows!, WindowsNT!
   path = "C:\excel\excel.xls"
CASE ELSE
   RETURN
END CHOOSE

Run(path)
```

GetFileOpenName

Description

Displays the system's Open File dialog and allows the user to select a file or enter a filename. If you specify a DOS-style file extension and the user enters a filename with no extension, PowerBuilder appends the default extension to the filename. If you specify a file mask to act as a filter, PowerBuilder displays only files that match the mask.

Syntax

GetFileOpenName (*title, pathname, filename* {,*extension* {, *filter* } })

Parameter	Description
title	A string whose value is the title of the dialog.
pathname	A string variable in which you want to store the returned path and filename.
filename	A string variable in which you want to store the returned filename.
extension (optional)	A string whose value is a 1- to 3-character default file extension. The default is no extension.
filter (optional)	A string whose value is a text description of the files to include in the listbox and the file mask that you want to use to select the displayed files (for example, *.* or *.exe). The format for *filter* is: `description,*.ext` The default is: "All Files (*.*),*.*"

Return value

Integer. Returns *1* if it succeeds, *0* if the user clicks the Cancel button or Windows cancels the display, and *-1* if an error occurs.

Usage

You use the *filter* argument to limit the types of files displayed in the list box and to let the user know what those limits are. For example, to display the description Text Files (*.TXT) and only files with the extension .TXT, specify the following for *filter*:

```
"Text Files (*.TXT),*.TXT"
```

To specify more than one file extension in *filter*, enter multiple descriptions and extension combinations and separate them with commas. For example:

```
"PIF files, *.PIF, Batch files, *.BAT"
```

Tip
Use the FileOpen function to open a selected file.

Platform information
On the Macintosh, the function processes filenames, extensions, and filters as it does in Windows.

Example

The following example displays the Open File window and if GetFileOpenName is successful, opens the file the user selects. The file types are TXT and DOC:

```
string docname, named
integer value

value = GetFileOpenName("Select File",  &
   + "docname, named, "DOC",  &
   + "Text Files (*.TXT),*.TXT,"  &
   + "Doc Files (*.DOC),*.DOC")

IF value = 1 THEN FileOpen(docname)
```

See also

DirList
DirSelect
GetFileSaveName

GetFileSaveName

Description

Displays the system's Save File dialog box with the specified filename displayed in the File name box. The user can enter a filename or select a file from the grayed list. If you specify a DOS-style extension and the user enters a filename with no extension, PowerBuilder appends the default extension to the filename. If you specify a file mask to act as a filter, PowerBuilder displays only files that match the mask.

Syntax

GetFileSaveName (*title*, *pathname*, *rfilename* {, *extension* {, *filter* }})

Parameter	Description
title	A string whose value is the title of the dialog box.
pathname	A string variable whose value is the default filename and which will store the returned path and filename. The default filename is displayed in the File name box, but the user can specify another name.
rfilename	A string variable in which you want to store the returned filename.
extension (optional)	A string whose value is a 1- to 3-character default file extension. The default is no extension.

Parameter	Description
filter (optional)	A string whose value is the description of the displayed files and the file extension that you want use to select the displayed files (the filter). The format for *filter* is: *description*, *.*ext* The default is: "All Files (*.*),*.*"

Return value

Integer. Returns *1* if it succeeds, *0* if the user clicks the Cancel button or Windows cancels the display, and *-1* if an error occurs.

Usage

See the GetFileOpenName function for usage notes on the *filter* argument.

Platform information
On the Macintosh, the function processes filenames, extensions, and filters as it does in Windows.

Example

These statements display the Save File window and list the files with the extension .TXT, then perform some processing if GetFileSaveName is successful. The title of the window is Select File and the file type is TXT:

```
string docname, named
integer value

value = GetFileSaveName("Select File",  &
   docname, named, "DOC",  &
   "Text Files (*.TXT),*.TXT," +  &
   " Doc Files (*.DOC), *.DOC")
IF value = 1 THEN ...
```

See also

GetFileOpenName
DirList
DirSelect

GetFirstSheet

Description

Obtains the top sheet in the MDI frame, which may or may not be active.

Applies to

MDI frame windows

Syntax

mdiframewindow.**GetFirstSheet** ( )

Parameter	Description
mdiframewindow	The MDI frame window for which you want the top sheet

Return value

Window. Returns the first (top) sheet in the MDI frame. If no sheet is open in the frame, GetFirstSheet returns an invalid value.

Usage

To cycle through the open sheets in a frame, use GetFirstSheet and GetNextSheet. Do not use these functions in combination with GetActiveSheet.

> **Did GetFirstSheet return a valid window?**
> Use the IsValid function to find out if the return value is valid. If it is not, then no sheet is open.

Example

This script for a MenuItem returns the top sheet in the MDI frame:

```
window wSheet
string wName
wSheet = ParentWindow.GetFirstSheet( )
IF IsValid(wSheet) THEN
      // There is an open sheet
      wName = wsheet.ClassName( )
      MessageBox("First Sheet is", wName)
END IF
```

See also

GetNextSheet
IsValid

GetFocus

Description

Determines the control that currently has focus.

Syntax

GetFocus ( )

Return value

GraphicObject. Returns the control that currently has focus. Returns a null control reference if an error occurs.

Example

These statements set which_control equal to the data type of the control that currently has focus, and then set text_value to the text attribute of the control:

```
GraphicObject which_control
SingleLineEdit sle_which
CommandButton cb_which
string text_value

which_control = GetFocus( )

CHOOSE CASE TypeOf(which_control)

CASE CommandButton!
   cb_which = which_control
   text_value = cb_which.Text

CASE SingleLineEdit!
   sle_which = which_control
   text_value = sle_which.Text

CASE ELSE
   text_value = ""
END CHOOSE
```

Note

Not all controls have a Text attribute (for example, the DataWindow control does not). For those controls, you could assign text to the Tag attribute, which is available for all controls, and access the Tag attribute instead of the Text attribute that is shown in this example.

See also

SetFocus

GetFormat

Description

Obtains the display format assigned to a column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetFormat** (*column*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the display format of a column.
column	The column for which you want the display format. *Column* can be a column number (integer) or a column name (string).

Return value

String. Returns the display format specification for *column* in *datawindowname*. If an error occurs, GetFormat returns the empty string ("").

Usage

If you want to temporarily change the display format of a column, you can use GetFormat to save the current format.

Example

These statements save the format of column salary of dw_employee before changing it to a new format:

```
string OldFormat, NewFormat = "$##,###.00"
OldFormat = dw_employee.GetFormat("salary")
dw_employee.SetFormat("salary", NewFormat)
```

See also

SetFormat

GetItemDate

Description

Gets data whose type is Date from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemDate** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the date data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must be date. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

Date. Returns NULL if the column value is NULL. Returns *1900-01-01* if an error occurs.

Usage

Use GetItemDate when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function; in this case date.

Examples

These statements set HireDate to the current Date data in the third row of the primary buffer in the column named first_day of dw_employee:

```
Date HireDate
HireDate = dw_employee.GetItemDate(3,  &
   "first_day")
```

These statements set HireDate to the current Date data in the third row of the filter buffer in the column named first_day of dw_employee:

```
Date HireDate
HireDate = dw_employee.GetItemDate(3,  &
   "first_day", Filter!)
```

These statements set HireDate to original Date data in the third row of the primary buffer in the column named hdate of dw_employee:

```
Date HireDate
HireDate = dw_employee.GetItemDate(3,  &
   "hdate", Primary!, TRUE)
```

See also

GetItemDateTime
GetItemDecimal
GetItemNumber
GetItemString
GetItemTime
GetText
SetItem
SetText

GetItemDateTime

Description

Gets data whose type is DateTime from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemDateTime** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the DateTime data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must be DateTime. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

DateTime. Returns NULL if the column value is NULL. Returns *1900-01-01 00:00:00.000000* if an error occurs.

Usage

Use GetItemDateTime when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function; in this case DateTime.

Examples

These statements set AsOf to the current DateTime data in the primary buffer for row 3 of the column named start_dt in the DataWindow dw_emp:

```
DateTime AsOf
AsOf = dw_emp.GetItemDateTime(3, "start_dt")
```

These statements set AsOf to the current DateTime data in the delete buffer for row 3 of the end_dt column of dw_emp:

```
DateTime AsOf
AsOf = dw_emp.GetItemDateTime(3, "end_dt", Delete!)
```

These statements set AsOf to the original DateTime data in the primary buffer for row 3 of the end_dt column of dw_emp:

```
DateTime AsOf
AsOf = dw_emp.GetItemDateTime(3, "end_dt", &
      Primary!, TRUE)
```

See also

GetItemDate
GetItemDecimal
GetItemNumber
GetItemString
GetItemTime
SetItem

GetItemDecimal

Description

Gets data whose type is Decimal from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemDecimal** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow Control in which you want to obtain the decimal data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must be one of a decimal data type. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

Decimal. Returns NULL if the column value is NULL. Returns *0* if an error occurs.

Usage

Use GetItemDecimal when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function, in this case a decimal data type.

Examples

These statements set salary_amt to the current decimal data in the primary buffer for row 4 of the column named emp_salary of dw_employee:

```
decimal salary_amt
salary_amt =  &
   dw_employee.GetItemDecimal(4, "emp_salary")
```

These statements set salary_amt to the current decimal data in the filter buffer for row 4 of the column named emp_salary of dw_employee:

```
decimal salary_amt
salary_amt = dw_employee.GetItemDecimal(4,  &
      "emp_salary", Filter!)
```

These statements set salary_amt to the original decimal data in the primary buffer for row 4 of the column named emp_salary of dw_employee:

```
decimal salary_amt
salary_amt = dw_employee.GetItemDecimal(4,  &
      "emp_salary", Primary!, TRUE)
```

See also

GetItemDate
GetItemDateTime
GetItemNumber
GetItemString
GetItemTime
SetItem

GetItemNumber

Description

Gets numeric data from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemNumber** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the numeric data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must be one of a numeric data type. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

A numeric data type (decimal, double, integer, long, or real). Returns NULL if the column value is NULL. Returns *0* if an error occurs.

Usage

Use GetItemNumber when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function; in this case a numeric data type.

Examples

These statements set EmpNbr to the current numeric data in the primary buffer for row 4 of the column named emp_nbr in dw_employee:

```
integer EmpNbr
EmpNbr = dw_employee.GetItemNumber(4, "emp_nbr")
```

These statements set EmpNbr to the current numeric data in the filter buffer for row 4 of the column named salary of dw_employee:

```
integer EmpNbr
EmpNbr = dw_employee.GetItemNumber(4,  &
   "salary", Filter!)
```

These statements set EmpNbr to the original numeric data in the primary buffer for row 4 of the column named salary of dw_Employee:

```
integer EmpNbr
EmpNbr = dw_Employee.GetItemNumber(4,  &
   "salary", Primary!, TRUE)
```

See also

GetItemDate
GetItemDateTime
GetItemDecimal
GetItemString
GetItemTime
SetItem

GetItemStatus

Description

Reports the modification status of a row or a column within a row. The modification status determines the type of SQL statement the Update function will generate for the row or column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemStatus** (*row, column, dwbuffer*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the status of a row or a column in a row.
row	A long identifying the row for which you want the status.
column	The column for which you want the status. *Column* can be a column number (integer) or a column name (string). Specify 0 to get the status of the whole row.
dwbuffer	A value of the dwBuffer enumerated data type identifying the DataWindow buffer that contains the row: ♦ PRIMARY! — The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow object). ♦ FILTER! — The data in the filter buffer (data that was filtered out).

Return value

A value of the dwItemStatus enumerated data type. Returns the status of the item at *row*, *column* of *datawindowname* in *dwbuffer*. If column is *0*, GetItemStatus returns the status of *row*.

Usage

The values of the dwItemStatus enumerated data type are:

- NotModified! — The information in the row or column is unchanged from what was retrieved.
- DataModified! — The information in the column or one of the columns in the row has changed since it was retrieved.

- New! — The row is new but no values have been specified for its columns. (Applies to rows only, not to individual columns.)
- NewModified! — The row is new, and values have been assigned to its columns. In addition to changes caused by user entry or the SetItem function, a new row gets the status NewModified! when one of its columns has a default value. (Applies to rows only, not to individual columns.)

Use GetItemStatus to understand what SQL statements will be generated for new and changed information when you update the database.

For information in the primary and filter buffers, Update generates an INSERT statement for rows with NewModified! status. It generates an UPDATE statement for rows with DataModified! status. Only columns with DataModified! status are included in the UPDATE statement.

For rows in the delete buffer, Update does not generate a DELETE statement for rows with New! or NewModified! status.

Examples

These statements store in the variable l_status the status of the column named emp_status in row 5 in the primary buffer of dw_1:

```
dwItemStatus l_status
l_status = &
   dw_1.GetItemStatus(5, "emp_status", Primary!)
```

These statements store in the variable l_status the status of the column named Salary in the current row in the primary buffer of dw_emp:

```
dwItemStatus l_status
l_status = dw_emp.GetItemStatus(dw_emp.GetRow(), &
   "Salary", Primary!)
```

See also

GetNextModified
SetItemStatus

GetItemString

Description

Gets data whose type is String from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemString** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the string data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must string. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

String. Returns NULL if the column value is NULL. Returns the empty string ("") if an error occurs.

Usage

Use GetItemString when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function, in this case String.

Examples

These statements set LName to the current string in the primary buffer for row 3 of in the column named emp_name in the DataWindow dw_employee:

```
String LName
LName = dw_employee.GetItemString(3, "emp_name")
```

These statements set LName to the current string in the delete buffer for row 3 of the column named emp_name of dw_employee:

```
String LName
LName = dw_employee.GetItemString(3,  &
   "emp_name", Delete!)
```

The following statements set LName to the original string in the delete buffer for row 3 of the column named emp_name of dw_employee:

```
String LName
LName = dw_employee.GetItemString(3,  &
   "emp_name", Delete!, TRUE)
```

See also

GetItemDate
GetItemDateTime
GetItemDecimal
GetItemNumber
GetItemTime
GetText
SetItem
SetText

GetItemTime

Description

Gets data whose type is Time from the specified buffer of a DataWindow control. You can obtain the data that was originally retrieved and stored in the database from the original buffer, as well as the current value in the primary, delete, or filter buffers.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetItemTime** (*row, column* &
{, *dwbuffer, originalvalue* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the time data contained in a specific row and column.
row	A long identifying the row location of the data.
column	The column location of the data. The data type of the column must time. *Column* can be a column number (integer) or a column name (string). **Tip** To get the contents of a computed field, specify the name of the computed field for *column*. Computed fields do not have numbers.
dwbuffer (optional)	A value of the dwBuffer enumerated data type identifying the DataWindow buffer from which you want to get the data: ♦ PRIMARY! — (Default) The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).
originalvalue (optional)	A boolean indicating whether you want the original or current values for *row* and *column*: ♦ TRUE — Return the original values, that is, the values initially retrieved from the database. ♦ FALSE — (Default) Return the current values. If you specify *dwbuffer*, you must also specify *originalvalue*.

Return value

Time. Returns NULL if the column value is NULL. Returns *00:00:00.000000* if an error occurs.

Usage

Use GetItemTime when you want to get information from the DataWindow's buffers. When you want to find out what the user entered in the current column before that data is accepted, use GetText.

To access a row in the original buffer, specify the buffer that the row currently occupies (primary, delete, or filter) and the number of the row in that buffer. When you specify TRUE for *originalvalue*, the function gets the original data for that row from the original buffer.

> **Note**
> An execution error occurs when the data type of the DataWindow column does not match the data type of the function, in this case time.

Examples

These statements set Start to the current Time data in the primary buffer for row 3 of the column named title in dw_employee:

```
Time Start
Start = dw_employee.GetItemTime(3, "title")
```

These statements set Start to the current Time data in the filter buffer for row 3 of the column named start_time of dw_employee:

```
Time Start
Start = dw_employee.GetItemTime(3,  &
   "start_time", Filter!)
```

These statements set Start to the original Time data in the primary buffer for row 3 of the column named start_time of dw_employee:

```
Time Start
Start = dw_employee.GetItemTime(3,  &
   "start_time", Primary!, TRUE)
```

See also

GetItemDate
GetItemDateTime
GetItemDecimal
GetItemNumber
GetItemString
GetText
SetItem
SetText

GetMessageText

Description

Obtains the message text generated by a crosstab DataWindow object in a DataWindow control. Only crosstab DataWindows generate messages.

Applies to

DataWindow controls

Syntax

datawindowname.**GetMessageText** ( )

Parameter	Description
datawindowname	The name of the DataWindow control for which you want the message text

Return value

String. Returns the text of the message generated by *datawindowname*. If there is no text or an error occurs, GetMessageText returns the empty string ("").

Usage

To use GetMessageText, you must first define a user-defined event for the event ID pbm_dwnmessagetext and then call this function in the script for that event.

Typical messages are "Retrieving data" and "Building crosstab."

Example

This statement is part of a script for a user-defined event with the ID pbm_dwmessagetext. The style of the DataWindow object in the DataWindow control is crosstab. The statement sets the MicroHelp of the MDI frame window w_crosstab:

```
w_crosstab.SetMicroHelp(This.GetMessageText( ))
```

GetNextModified

Description

Reports the next row that has been modified in the specified buffer.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetNextModified** (*row, dwbuffer*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to locate the modified row.
row	A long identifying the row location after which you want to locate the modified row. To search from the beginning, specify 0.
dwbuffer	A value of the dwBuffer enumerated data type identifying the DataWindow buffer in which you want to locate the modified row: ♦ PRIMARY! — The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow). ♦ FILTER! — The data in the filter buffer (data that was filtered out).

Return value

Long. Returns the number of the first row that was modified after *row* in *dwbuffer* in *datawindowname*. Returns *0* if there are no modified rows after the specified row.

Usage

PowerBuilder stores the update status of rows and columns in the data window. The status settings indicate whether a row or column is new or has been modified. GetNextModified reports rows with the status NewModified! and DataModified!. See GetItemStatus and SetItemStatus for more information on the status for rows and columns.

Using GetNextModified on the deleted buffer will return rows that have been modified and then deleted. The DeletedCount function will report the total number of deleted rows.

GetNextModified begins searching in the row after the value you specify in *row*. This is different from the behavior of Find, FindGroupChange, and FindRequired, which beginning searching in the row you specify.

Example

These statements count the number or rows that were modified in the primary buffer for dw_status and then display a message reporting the number modified:

```
integer rc
long NbrRows, row = 0, count = 0
dwItemStatus status

dw_status.AcceptText()
NbrRows = dw_status.RowCount( )
DO WHILE row <= NbrRows
   row = dw_status.GetNextModified(row, Primary!)
   IF row > 0 THEN
   count = count + 1
  ELSE
   row = NbrRows + 1
  END IF
LOOP
MessageBox("Modified Count",  &
   String(count) &
   + " rows were modified.")
```

Note
You can use the function ModifiedCount to find out the total number of modified rows in the primary and filter buffers.

See also

DeletedCount
FindRequired
GetNextModified
ModifiedCount
SetItemStatus

GetNextSheet

Description

Obtains the sheet that is behind the specified sheet in the MDI frame.

Applies to

MDI frame windows

Syntax

mdiframewindow.**GetNextSheet** (*sheet*)

Parameter	Description
mdiframewindow	The MDI frame window in which you want the next sheet
sheet	The sheet for which you want the sheet that is behind it

Return value

Window. Returns the sheet that is behind *sheet* in the MDI frame. If there is no sheet behind *sheet*, GetNextSheet returns an invalid value.

Usage

To cycle through the open sheets in a frame, use GetFirstSheet to get the front sheet and GetNextSheet one or more times to get the rest of the sheets. Test each return value with IsValid to see if you have reached the last sheet.

Do not use GetFirstSheet and GetNextSheet in combination with GetActiveSheet.

> **Did GetNextSheet return a valid window?**
> Use the IsValid function to find out if GetNextSheet returned a valid window. If there is no sheet behind the one you specified, the return value will not be valid.

Example

The following script for a MenuItem loops through the open sheets in front-to-back order and displays the names of the open sheets in the ListBox lb_sheets:

```
boolean bValid
window  wSheet

lb_sheets.Reset( )

wSheet = ParentWindow.GetFirstSheet()
IF IsValid(wSheet) THEN
   lb_sheets.AddItem(wSheet.Title)
   DO
      wSheet = ParentWindow.GetNextSheet(wSheet)
      bValid = IsValid (wSheet)
      IF bValid THEN lb_sheets.AddItem(wSheet.Title)
   LOOP WHILE bValid
END IF
```

See also

GetFirstSheet
IsValid

GetObjectAtPointer

Description

Reports the object and row number under the pointer. DataWindow objects include columns, labels, and other graphic objects, such as lines and bitmaps.

Applies to

DataWindow controls

Syntax

datawindowname.**GetObjectAtPointer** ( )

Parameter	Description
datawindowname	The name of the DataWindow control in which you want to obtain the object under the pointer

Return value

String. Returns the string whose value is the name of the object under the pointer, followed by a tab character and the row number. Returns the empty string ("") if an error occurs.

Usage

If the object doesn't have a name, neither a name or a row is reported. PowerBuilder gives columns and column labels names in the DataWindow painter. You can name other objects yourself in the DataWindow painter.

You can parse the return value by searching for the tab character ("~t" or ASCII 09). For sample code that parses the return value, see the Pos function.

For information on the rows associated with bands and therefore with objects in those bands, see GetBandAtPointer.

Example

These statements obtain the object under the pointer in the DataWindow dw_emp:

```
String dwobject
dwobject = dw_emp.GetObjectAtPointer()
```

Some possible return values are:

Return value	Meaning
salary~t23	The object named salary in row 23.
salary_h~t15	The object named salary_h, which is in the header. Row 15 is the first visible row below the header.

See also GetBandAtPointer

GetRemote

Description

Asks a DDE server application to provide data and stores that data in the specified variable. There are two ways of calling GetRemote, depending on the type of DDE connection you've established:

- When you are making a single DDE request of a server application (a cold link), use Syntax 1.
- When you have established a warm link by opening a channel to the server application, use Syntax 2. A warm link, with an open channel, is more efficient when you intend to make several DDE requests.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax 1

GetRemote (*location, target, applname, topicname*)

Parameter	Description
location	A string whose value is the location of the data you want returned from the DDE server application. The format of *location* depends on the particular DDE server application that will receive the message.
target	A string variable into which the returned data will be placed.
applname	A string whose value is the DDE name of the DDE server application. If another PowerBuilder application is the DDE server, this is the application name specified in its StartServerDDE function call.
topicname	A string identifying the data or the instance of the application you want to use with the command (for example, in Microsoft Excel, the topic name could be system or the name of an open spreadsheet). If another PowerBuilder application is the DDE server, this is the topic specified in its StartServerDDE function call.

Return value 1

Integer. Returns *1* if it succeeds and a negative integer if an error occurs. Values are:

- *-1* Link was not started
- *-2* Request denied

Syntax 2

GetRemote (*location, target, handle* {, *windowhandle* })

Parameter	Description
location	A string whose value is the location of the data you want returned. The format of the location depends on the DDE application that will receive the request.
target	A PowerBuilder string variable into which the returned data will be placed.
handle	A long that identifies the channel to the DDE server application. The OpenChannel function returns *handle* when you call it to open a DDE channel.
windowhandle (optional)	The handle to the window that is acting as the DDE client. Specify this parameter to control which window the data is returned to when you have more than one open window.

Return value 2

Integer. Returns *1* if it succeeds and a negative integer if an error occurs. Values are:

- *-1* Link was not started
- *-2* Request denied
- *-9* handle is NULL

Usage

When using DDE, your PowerBuilder application must have an open window, which will be the client window. For Syntax 1, the active window is the DDE client window. For Syntax 2, you can specify a different client window with the *windowhandle* argument.

Before using Syntax 2 of GetRemote, call OpenChannel to establish a DDE channel.

For more information about DDE channels and warm and cold links, see the ExecRemote function.

Examples

These statements ask Microsoft Excel to get the data in row 1 column 2 of a worksheet called PROFIT.XLS and put it in a PowerBuilder string called ls_ProfData. Here, using Syntax 1, the single GetRemote call establishes a cold link, gets the data, and ends the link:

```
string ls_ProfData
GetRemote("R1C2", ls_ProfData,  &
   "Excel", "PROFIT.XLS")
```

Syntax 2

These statements ask the channel identified by handle (a Microsoft Excel worksheet) to get the data in row 1 column 2 and save it in a PowerBuilder string called ls_ProfData. Here, using Syntax 2, GetRemote utilizes the warm link established by the OpenChannel function:

```
String ls_ProfData
long handle

handle = OpenChannel("Excel", "REGION.XLS")
. . .
GetRemote("R1C2", ls_ProfData, handle)
. . .
CloseChannel(handle)
```

The following example is similar to the previous one. However, it specifically associates the DDE channel with the window w_rpt:

```
String ls_ProfData
long handle

handle = OpenChannel("Excel", "REGION.XLS", &
      Handle(w_rpt))
. . .
GetRemote("R1C2", ls_ProfData,  &
      handle, Handle(w_rpt))
. . .
CloseChannel(handle, Handle(w_rpt))
```

See also

CloseChannel
ExecRemote
OpenChannel
SetRemote

GetRow

Description

Reports the number of the current row in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetRow** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or the child DataWindow for which you want the number of the current row

Return value

Long. Returns the number of the current row in *datawindowname*. Returns *0* if no row is current and *-1* if an error occurs.

> **Tip**
> The current row is not always a row displayed on the screen. For example, if the cursor is on row 7 column 2 and the user uses the scroll bar to scroll to row 50, the current row remains row 7 unless the user clicks row 50.

Example

This statement returns the number of the current row in dw_Employee:

```
dw_employee.GetRow( )
```

See also

GetColumn
SetColumn
SetRow
GetRow in Chapter 2, "DataWindow Painter Functions"

GetSelectedRow

Description

Reports the number of the next highlighted row after a specified row in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetSelectedRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the number of a selected row
row	A long identifying the location of the row after which you want to search for the next selected row

Return value

Long. Returns the number of the first row that is selected after *row* in *datawindowname*. Returns *0* if no row is selected after the specified row.

Usage

Rows are not automatically selected, that is, highlighted, when they become current. You can select a row by calling the SelectRow function.

GetSelectedRow begins it search *after* the specified row. It doesn't matter whether *row* itself is selected.

Examples

This statement returns the number of the first row that is selected in dw_Employee:

```
dw_employee.GetSelectedRow(0)
```

This statement returns the number of the first row that is selected beginning with row 25 in dw_Employee:

```
dw_employee.GetSelectedRow(25)
```

See also

SelectRow

GetSeriesStyle

Description

Finds out the appearance of a series in a graph. The appearance settings for individual data points can override the series settings, so the values obtained from GetSeriesStyle may not reflect the current state of the graph.

There are several syntaxes:

- To get the series' colors, use Syntax 1.
- To get the series' line style and width, use Syntax 2.
- To get the fill pattern or symbol for the series, use Syntax 3.
- To find out if the series is an overlay, that is, a series shown as a line on top of another graph type, use Syntax 4.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax 1

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *colortype*, *colorvariable*)

Parameter	Description
controlname	The name of the graph in which you want to obtain the color of a series, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the color of a series
seriesname	A string whose value is the name of the series for which you want the color
colortype	A value of the grColorType enumerated data type specifying the aspect of the series for which you want the color: ♦ Foreground! — Text color ♦ Background! — Background color ♦ LineColor! — Line color ♦ Shade! — Shade (for graphs that are three-dimensional or have solid data markers)
colorvariable	A long variable in which you want to store the color's RGB value

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores in *colorvariable* the RGB value of the specified series and item.

Syntax 2

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *linestyle*, *linewidth*)

Parameter	Description
controlname	The name of the graph for which you want the line style and width for a series in a graph, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the line style information.
seriesname	A string whose value is the name of the series for which you want the line style information.
linestyle	A variable of type LineStyle in which you want to store the line style of *seriesname*.
linewidth	An integer variable in which you want to store the line width for *seriesname*. The width is measured in pixels.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores in *linestyle* a value of the LineStyle enumerated data type and in *linewidth* the width of the line used for the specified series.

Syntax 3

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *enumvariable*)

Parameter	Description
controlname	The name of the graph for which you want the style information for a series in a graph, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the style information
seriesname	A string whose value is the name of the series for which you want the style information
enumvariable	The variable in which you want to store the style information. You can specify a FillPattern or grSymbolType variable. The style information that GetSeriesStyle stores depends on the variable type.

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores in *enumvariable* a value of the appropriate enumerated data type for the fill pattern or symbol used for the specified series.

Syntax 4

controlname.**GetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *overlayindicator*)

Parameter	Description
controlname	The name of the graph for which you want the overlay status of a series in a graph, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the overlay status
seriesname	A string whose value is the name of the series for which you want the overlay status
overlayindicator	A boolean variable in which you want to store a value indicating whether the series is an overlay. GetSeriesStyle sets *overlayindicator* to TRUE if the series is an overlay and FALSE if it is not.

Return value 4

Integer. Returns *1* if it succeeds and *-1* if an error occurs. Stores in *overlayindicator* TRUE if the specified series is an overlay and FALSE if it is not.

Usage

GetSeriesStyle provides information about a series. The data points in the series can have their own style settings. Use SetSeriesStyle to change the style values for a series. Use GetDataStyle get style information for a data point and SetDataStyle to override series settings and set style information for individual data points.

The graph stores style information for attributes that don't apply to the current graph type. For example, you can find out the fill pattern for a data point or a series in a two-dimensional line graph, but that fill pattern will not be visible.

See SetSeriesStyle for a list of the enumerated data type values that GetSeriesStyle stores in *enumvariable*.

Examples

These statements store in the variable color_nbr the text (foreground) color used for a series in the graph gr_emp_data. The series name is the text in the SingleLineEdit sle_series:

```
long color_nbr
gr_emp_data.GetSeriesStyle(sle_series.Text, &
   Foreground!, color_nbr)
```

These statements store in the variable color_nbr the background color used for the series PCs in the graph gr_computers in the DataWindow control dw_equipment:

```
long color_nbr
// Get the color.
dw_equipment.GetSeriesStyle("gr_computers",  &
   "PCs", Background!, color_nbr)
```

These statements store the color for the series under the mouse pointer in the graph gr_product_data in line_color:

```
string SeriesName
integer SeriesNbr, Data_Point
long line_color
grObjectType MouseHit

MouseHit = ObjectAtPointer(SeriesNbr, Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
      gr_product_data.SeriesName(SeriesNbr)

   gr_product_data.GetSeriesStyle(SeriesName, &
      LineColor!, line_color)
END IF
```

Syntax 2

These statements store in the variables line_style and line_width the line style and width for the series under the mouse pointer in the graph gr_product_data:

```
string  SeriesName
integer SeriesNbr, Data_Point, line_width
LineStyle line_style
grObjectType MouseHit

MouseHit = ObjectAtPointer(SeriesNbr,  Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
      gr_product_data.SeriesName(SeriesNbr)

   gr_product_data.GetSeriesStyle(SeriesName, &
      line_style, line_width)
END IF
```

Syntax 3

These statements store in the variable data_pattern the fill pattern for the series under the mouse pointer in the graph gr_product_data:

```
string  SeriesName
integer SeriesNbr, Data_Point
FillPattern data_pattern
grObjectType MouseHit

MouseHit = ObjectAtPointer(SeriesNbr,  Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
      gr_product_data.SeriesName(SeriesNbr)

   gr_product_data.GetSeriesStyle(SeriesName, &
      data_pattern)
END IF
```

This example stores in the variable data_pattern the fill pattern for the series under the pointer in the graph gr_depts in the DataWindow control dw_employees. It then sets the fill pattern for the series Total Salary in the graph gr_dept_data to that pattern:

```
string SeriesName
integer SeriesNbr, Data_Point
FillPattern data_pattern
grObjectType MouseHit

MouseHit =  &
   ObjectAtPointer("gr_depts", SeriesNbr,  &
      Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
     dw_employees.SeriesName("gr_depts", SeriesNbr)

   dw_employees.GetSeriesStyle("gr_depts", &
      SeriesName, data_pattern)

   gr_dept_data.SetSeriesStyle("Total Salary",  &
      data_pattern)
END IF
```

In the examples above, you can change the data type of data_pattern, the variable specified as the last argument, to find out the symbol type.

Syntax 4

These statements find out whether a series in the graph gr_emp_data is an overlay. The series name is the text in the SingleLineEdit sle_series:

```
boolean is_overlay
gr_emp_data.GetSeriesStyle(sle_series.Text,  &
   is_overlay)
```

See also

AddSeries
GetDataStyle
FindSeries
SetDataStyle
SetSeriesStyle

GetSQLPreview

Description

Reports the SQL statement that the DataWindow control is currently submitting to the database.

> **Note**
> Call this function *only* in a script for the DBError event or the SQLPreview event.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetSQLPreview ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to obtain the SQL statement the DataWindow control is submitting currently to the database server

Return value

String. Returns the current SQL statement for *datawindowname*. Returns the empty string ("") if an error occurs.

Usage

You can call GetSQLPreview in the SQLPreview event script to obtain the SQL statement being executed. In that script, you can modify the returned SQL statement and call SetSQLPreview to change the SQL to be executed. You can also write the SQL statement to a log file.

GetSQLPreview and binding

When binding is enabled for your database, the SQL returned in the GetSQLPreview event may not be complete—the input arguments are not replaced with the actual values. For example, when binding is enabled, GetSQLPreview might return the following SQL statement:

```
INSERT INTO "cust_order" ( "ordnum", "custnum",
"duedate", "balance" ) VALUES ( ?, ?, ?, ? )
```

When binding is disabled, it returns:

```
INSERT INTO "cust_order" ( "ordnum", "balance",
"duedate", "custnum" ) VALUES ( '12345', 900,
'3/1/94', '111' )
```

If you require the complete SQL statement for logging purposes, you should disable binding in your DBMS. See *Connecting to Your Database* for more information about binding.

Example

In the script for the SQLPreview event in dw_status, this statement displays the current SQL statement for dw_status in a MultiLineEdit called mle_sql:

```
mle_sql.Text = dw_status.GetSQLPreview( )
```

See also

SetSQLPreview

GetSQLSelect

Description

Reports the SQL SELECT statement associated with a DataWindow if its data source is one that accesses an SQL database, for example, SQL Select, Quick Select, or Query.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetSQLSelect ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to obtain the current SELECT statement.

Return value

String. Returns the current SQL SELECT statement for *datawindowname*. GetSQLSelect returns the empty string ("") if it cannot return the statement.

Usage

When you want to change the SQL SELECT statement for a DataWindow during execution, you can use GetSQLSelect to save the current SELECT statement before making the change.

When you define a DataWindow, PowerBuilder stores a PowerBuilder SELECT statement (PBSELECT) with the DataWindow. If a database is connected and SetTransObject has been called for the DataWindow, then GetSQLSelect returns the SQL SELECT statement. Otherwise, GetSQLSelect returns the PBSELECT statement.

You can also use Describe to obtain the SQL SELECT statement. (See the Table.Select attribute in Appendix A.)

Example

These statements save the SELECT statement for dw_emp before it is temporarily modified:

```
string old_select, new_select, where_clause

old_select = dw_emp.GetSQLSelect( )
where_clause = ...

// Add the new where clause to old_select
new_select = old_select + where_clause
dw_emp.SetSQLSelect(new_select)
```

See also

SetSQLSelect
Appendix A, "DataWindow Object Attributes"

GetText

Description

Obtains the value in the edit control over the current row and column. When the user changes a value in a DataWindow, it is available in the edit control before it is accepted into the column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetText** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the text of the current row and column

Return value

String. Returns the value in the edit box over the current row and column in *datawindowname*. The value may or may not have been accepted into the row and column. Returns the empty string ("") if no column is currently selected in *datawindowname*.

Usage

The values in the rows and columns of a DataWindow are items in the DataWindow's buffer. When a user edits a value in a row and column, the item value is transferred, as text, to the edit control, where the user can change the value. When the user leaves the column or when a script calls AcceptText, the text in the edit control is accepted into the column and becomes the value of the item in the buffer.

Call GetText in the script for the ItemChanged or ItemError event to check the value entered in the edit control over the current row and column before allowing it to be accepted into the column.

To obtain the value stored in the DataWindow's buffer for the row and column, use the GetItem function that corresponds with the data type of the column.

Example

These statements in the script for the ItemChanged event for dw_employee return the text in dw_employee:

```
string LName
LName = dw_employee.GetText( )
```

See also

SetText
GetText in Chapter 2, "DataWindow Painter Functions"

GetTrans

Description

Gets the values for the DataWindow control's internal transaction object and stores them in the programmer-specified transaction object.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetTrans** (*transaction*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow from which you want to get the internal transaction object values.
transaction	The name of the transaction object into which you want to put the values.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

The SetTrans function (not the SetTransObject function) sets the internal transaction object. If you have not called SetTrans, GetTrans will fail.

Use GetTrans when you want to get the values for the transaction object in order to modify them, as shown in the last example.

If you are using SetTransObject, which specifies transaction information via a programmer-specified transaction object, GetTrans will not report information about the programmer-specified transaction object currently in effect. (SetTransObject is the recommended connection method because it gives better application performance. See SetTrans and SetTransObject for more information.)

Examples

This example puts the values in the internal transaction object for dw_employee into the programmer-specified transaction object named object1:

```
transaction object1
object1 = CREATE transaction
dw_employee.GetTrans(object1)
```

The following statement puts the values in the internal transaction object for dw_employee into the default transaction object (SQLCA):

```
dw_employee.GetTrans(SQLCA)
```

The following statements change the database type and password of dw_employee. The first two statements create the transaction object emp_TransObj. The next two statements use the SetTrans function to set the values of SQLCA, and then use the GetTrans function to store the values of the current transaction object for dw_employee in emp_TransObj. The last two statements change the database type and password and then the SetTrans function puts the revised values in the transaction object for dw_employee:

```
// Name the transaction object.
transaction emp_TransObj

// Create the transaction object.
emp_TransObj = CREATE transaction

// Set the internal transaction object.
dw_employee.SetTrans(SQLCA)

// Fill the new transaction object with original
// values from SQLCA.
dw_employee.GetTrans(emp_TransObj)

// Put revised values into the new transaction
// object.
// Change the database type.
emp_TransObj.DBMS = "Sybase"

// Change the password.
emp_TransObj.LogPass = "cam2"

// Associate the new transaction object with
// dw_employee, replacing SQLCA.
dw_employee.SetTrans(emp_TransObj)
```

See also

SetTrans

GetUpdateStatus

Description

Reports the row number and buffer of the row that is currently being updated in the database. When called because of an error, GetUpdateStatus reports the row that caused the error.

> **Note**
> Call this function *only* in the script for the DBError event or the SQLPreview event.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GetUpdateStatus** (*row, dwbuffer*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow.
row	A long variable that will store the number of the row that will be updated or for which an update was attempted.
dwbuffer	A dwBuffer variable that will store a value of the dwBuffer enumerated data value for the DataWindow buffer that contains the row that will be updated. Possible values are: ♦ PRIMARY! — The data in the primary buffer (the data that has not been deleted or filtered out). ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow object). ♦ FILTER! — The data in the filter buffer (data that was filtered out).

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The number and buffer of the row currently being updated are stored in *row* and *dwbuffer*.

Usage

Call GetUpdateStatus in the DBError event, which is triggered when a database error occurs, to find out which row caused the error.

If the row that caused the error is in the Filter buffer, you must unfilter it if you want the user to correct the problem.

Reported row number
The row number stored in *row* is the number of the row in the buffer, not the number the row had when it was retrieved into the DataWindow object.

Example

These statements in the script for the DBError event for a DataWindow control obtain the text of the error message, display a message box with the number of the row in which the error occurred and the error message, and then make the row with the error the current row.

Additional code in the IF statement considers the case of the bad row being in the filter or delete buffer. If the row is in the filter buffer, the script changes the filter so that the user can edit the row in the primary buffer. If the row is in the delete buffer, the message box displays a slightly different title:

```
long row_number, row_key
dwBuffer buffer_type
string message_text, message_title, old_filter

// Get the error message text and set the title
message_text = DBErrorMessage( )
message_title = "Database Error Updating Row"

// Get the row in which the error occurred
This.GetUpdateStatus(row_number, buffer_type)

IF buffer_type = Filter! THEN
   old_filter = This.Describe("DataWindow.Filter")
   row_key = This.GetItemNumber(row_number,  &
      "emp_id", Filter!, FALSE)

   This.SetFilter("(" + old_filter + ")" +  &
      "OR emp_id = " + String(row_key))
   This.Filter( )

   // Error row is now last row in primary buffer
   row_number = This.RowCount( )

ELSEIF buffer_type = Delete! THEN
   message_title = "Database Error Deleting Row"

END IF

// Display the location of the error and the error
// message.
MessageBox(message_title +  &
   String(row_number), message_text)
```

```
IF buffer_type <> Delete! THEN
   // Make the row with the error the current row.
   This.ScrollToRow(row_number)
END IF

// Set the action code for the DBError event to 1
// (do not display error message) because we've
// already displayed a message
This.SetActionCode(1)
```

See also GetItemStatus

GetValidate

Description Obtains the validation rule for a column in a DataWindow.

Applies to DataWindow controls and child DataWindows

Syntax *datawindowname*.**GetValidate** (*column*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to obtain the validation rule for a column.
column	The column for which you want the validation rule. *Column* can be a column number (integer) or a column name (string).

Return value String. Returns the validation rule for *column* in *datawindowname*. Returns the empty string ("") if no validation criterion is defined for the column.

Usage You can use GetValidate to save the current validation rule before calling SetValidate to change the rule temporarily.

Example These statements change the validation rule for column 7 in the DataWindow control dw_Employee to Rule2:

```
string Rule1, Rule2 = "Long(GetText()) > 15000"
Rule1 = dw_Employee.GetValidate(7)
dw_Employee.SetValidate(7, Rule2)
```

See also SetValidate

GetValue

Description Obtains the value of an item in a value list or code table associated with a column in a DataWindow.

Applies to DataWindow controls and child DataWindows

Syntax *datawindowname*.**GetValue** (*column*, *index*)

Parameter	Description
datawindowname	The name of the DataWindow control in which you want to obtain the value of an item in a value list or the code table.
column	The column for which you want the item. *Column* can be a column number (integer) or a column name (string).
index	The number of the item in the value list or the code table for the edit style.

Return value String. Returns the item identified by *index* in the value list or the code table associated with *column* of *datawindowname*. If the item has a display value that is not the actual value, GetValue returns a tab-separated string with the display value on the left of the tab and the code value on the right of the tab.

Returns the empty string ("") if the index is not valid or the column does not have a value list or code table.

Usage You can use GetValue to find out the values associated with the following edit styles: CheckBox, RadioButton, DropDownListBox, Edit Mask, and Edit. If the edit style has a code table, in which each value in the list has a display value and a data value, GetValue reports both values.

GetValue does not get values from a DropDownDataWindow code table.

For sample code that parses the return value when it is a pair of tab-separated values from a code table, see the Pos function.

Examples

If the value list for column 7 of dw_employee contains Full Time, Part Time, Retired, and Terminated, these statements return the value of item 3 (Retired):

```
string Status
Status = dw_employee.GetValue(7,3)
```

If the value list for the column named product of dw_employee is Widget~t1, Gadget~t2, the following statement returns Gadget~t2:

```
ls_pval = dw_employee.GetValue("product", 2)
```

See also

ClearValues
SetValue

GroupCalc

Description

Recalculates the breaks in the grouping levels in a data window.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**GroupCalc** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to recalculate breaks within the grouping levels

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use GroupCalc to force the DataWindow object to recalculate the breaks in the grouping levels after you have added or modified rows in a DataWindow.

GroupCalc does not sort the data before it recalculates the breaks. Therefore, unless you populated the DataWindow in a sorted order, call the Sort function to sort the data before you call GroupCalc.

Example

This code imports new rows from a file into the DataWindow dw_emp and then recalculates the group breaks for dw_emp:

```
ImportFile(dw_emp, "d:\employee.txt")
dw_emp.SetSort("1A")
dw_emp.Sort( )
dw_emp.GroupCalc( )
```

See also

Sort

Handle

Description

Obtains the Windows handle of a PowerBuilder object. You can get the handle of the application, a window, or a control, but not a drawing object.

> **Platform information**
> On the Macintosh, Handle does not return a handle that you can use with external Macintosh functions.

Syntax

Handle (*objectname* {, *previous* })

Parameter	Description
objectname	The name of the PowerBuilder object for which you want the handle. *Objectname* can be any PowerBuilder object, including an application or control, but cannot be a drawing object.
previous (optional)	A boolean indicating whether you want the handle of the previous instance of an application. Values are: ♦ FALSE — (Default) Return the handle of the current instance. ♦ TRUE — Return the handle of the previous instance.

Return value

Long. Returns the handle of *objectname*. If *objectname* is an application and *previous* is TRUE, Handle returns *0* if there is no previous instance.

If *objectname* cannot be referenced during execution, Handle returns *0* (for example, if *objectname* is a window and is not open).

Usage

Use Handle when you need an object handle as an argument to Windows Software Development Kit (SDK) functions or PowerBuilder's Send function.

Use IsValid instead of the Handle function to determine whether a window is open.

When you ask for the handle of the application, Handle returns 0 when you are using PowerBuilder's Run command. As far as Windows is concerned, your application does not have a handle when it is run from PowerBuilder. When you build and run an executable version of your application, the Handle function returns a valid handle for the application.

When you ask for the handle of a previous instance of the application, Handle returns 0 if no other instance is running. You can use Handle with this argument to prevent the user from running multiple instances of your application (see the examples).

Examples

This statement returns the handle to the window w_child:

```
Handle(w_child)
```

These statements use an external function in Windows called FlashWindow to change the title bar of a window to inactive and then return it to active. The external function declaration is:

```
function boolean flashwindow(uint hnd, boolean inst)
library "user.exe"
```

The code that flashes the window's title bar is:

```
integer nLoop     // Loop counter
long hWnd         // Handle to control

// Get the handle to a PowerBuilder window.
hWnd = Handle(Parent)

// Make the title bar flash 300 times.
FOR nLoop = 1 to 300
   FlashWindow (hWnd, TRUE)
NEXT

// Return the window to its original state.
FlashWindow (hWnd, FALSE)
```

This example from the script of the application's Open event checks whether the application is already running and, if so, prevents the application from being started another time. If not, it opens the application's window w_main:

```
IF Handle(This, TRUE) > 0 THEN
   MessageBox("Application Already Running",  &
      This.AppName + " is already running."  &
      + " You cannot start it again.")
   HALT CLOSE
ELSE
   Open(w_main)
END IF
```

See also

Send

Hide

Description

Makes an object or control invisible. Users cannot interact with an invisible object. It doesn't respond to any events, so the object is also, in effect, disabled.

Applies to

Any object

Syntax

objectname.**Hide** ( )

Parameter	Description
objectname	The name of the object or control you want to make invisible

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If the object you want to hide is already invisible, then Hide has no effect.

You cannot use Hide to hide a dropdown or cascading menu or any menu that has an MDI frame window as its parent window. Nor can you hide a window that has been opened as an MDI sheet.

You can use the Disable function to disable menu items, which displays them in the disabled color and makes them inactive.

To disable an object so that it doesn't respond to events, but is still visible, set its Enabled attribute.

Equivalent syntax

You can set an object's Visible attribute instead of calling Hide:

objectname.Visible = FALSE

This statement:

```
lb_Options.Visible = FALSE
```

is equivalent to:

```
lb_Options.Hide ( )
```

Examples

This statement hides the ListBox lb_options:

```
lb_options.Hide( )
```

In the script for a menu item, this statement hides the CommandButton cb_delete on the active sheet in the MDI frame w_mdi. The active sheets are of type w_sheet:

```
w_sheet w_active
w_active = w_mdi.GetActiveSheet()
IF IsValid(w_active) THEN w_active.cb_delete.Hide( )
```

See also

Show

Hour

Description

Obtains the hour in a time value. The hour is based on a 24-hour clock.

Syntax

Hour (*time*)

Parameter	Description
time	The time from which you want to obtain the hour

Return value

Integer. Returns an integer (00 to 23) whose value is the hour portion of *time*.

Examples

This statement returns the current hour:

```
Hour(Now( ))
```

This statement returns 19:

```
Hour(19:01:31)
```

See also

Minute
Now
Second
Hour in Chapter 2, "DataWindow Painter Functions"

Idle

Description

Sets a timer so that PowerBuilder triggers an Application Idle event when there has been no user activity for a specified number of seconds.

Syntax

Idle (*n*)

Parameter	Description
n	The number of seconds of user inactivity allowed before PowerBuilder triggers an Application Idle event. A value of 0 terminates Idle detection.

Return value

Integer. Returns *1* if it starts the timer, and *-1* if it cannot start the timer or *n* is 0 and the timer has not been started. Note that when the timer has been started and you change *n*, Idle does not start a new timer; it resets the current timer interval to the new number of seconds. The return value is usually not used.

Usage

Use Idle to shut off or restart an application when there is no user activity. This is often done for security reasons.

Idle starts a timer after each user activity (for example, a keystroke or a mouse click), and after *n* seconds of inactivity it triggers an Idle event. The Idle event script for an application typically closes some windows, logs off the database, and exits the application or calls the Restart function.

The timer is reset when any of the following activities occur:

- A mouse movement or mouse click in any window of the application
- Any keyboard activity when a window of the PowerBuilder application is current
- A mouse click or any mouse movement over the icon when a PowerBuilder application is minimized
- Any keyboard activity when the PowerBuilder application is minimized and is current (its name is highlighted)

Examples

This statement sends an Idle event after five minutes of inactivity:

```
Idle(300)
```

This statement turns off idle detection:

```
Idle(0)
```

The following example shows how to use the Idle event to stop the application and restart it after two minutes of inactivity. This is often used for computers that provide information in a public place.

Include in the script for the application's Open event:

```
Idle(120)   //Sends an Idle event after 2 minutes.
```

Include these statements in the script for the application's Idle event to terminate the application and then restart it:

```
//Statements to set the database to the desired
//state
. . .
Restart( )  //Restarts the application
```

See also

Restart
Timer

ImportClipboard

Description

Inserts data into a DataWindow control or graph control from tab-delimited data on the clipboard. There are two syntaxes:

- To import rows into a DataWindow control, use Syntax 1.
- To add new series to a graph control, use Syntax 2.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

datawindowname.**ImportClipboard**({ *startrow* {, *endrow* & {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow to which you want to copy data from the clipboard.
startrow (optional)	The number of the first row in the clipboard that you want to copy. If the first row on the clipboard contains headings that you want to skip, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row in the clipboard that you want to copy. The default is the rest of the rows.
startcolumn (optional)	The number of the first column in the clipboard that you want to copy. The default is 1.
endcolumn (optional)	The number of the last column in the clipboard that you want to copy. The default is the rest of the columns.
dwstartcolumn (optional)	The number of the first column in the DataWindow control that you want to receive data. The default is 1.

Return value

Long. Returns the number of rows that were imported if it succeeds and one of the following negative integers if an error occurs:

- *-3* Invalid argument
- *-4* Invalid input

Syntax 2

graphname.**ImportClipboard** ({ *startrow* {, *endrow* & {, *startcolumn* } } })

Parameter	Description
graphname	The name of the graph control to which you want to copy data from the clipboard.
startrow (optional)	The number of the first row in the clipboard that you want to copy. If the first row on the clipboard contains headings that you want to skip, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row in the clipboard that you want to copy. The default is the rest of the rows.
startcolumn (optional)	The number of the first column in the clipboard that you want to copy. The default is 1.

Return value 2

Long. Returns the number of rows that were imported if it succeeds and returns the following values if an error occurs:

- *0* End of file, too many rows
- *-2* Not enough columns
- *-3* Invalid argument
- *-4* Invalid input

Usage

The clipboard data must be formatted in tab-delimited columns.

For DataWindow controls, the data types and order of the DataWindow's columns must match the data on the clipboard.

The *startcolumn* and *endcolumn* arguments control the number of imported columns and the number of columns in the DataWindow that are affected. The *dwstartcolumn* argument specifies the first DataWindow column to be affected. The following formula calculates the last DataWindow to be affected:

dwstartcolumn + (*endcolumn* - *startcolumn*)

For graph controls, ImportClipboard only uses three columns and ignores other columns. Each row of data must contain three pieces of information. The information depends on the type of graph:

- For all graph types except scatter, the first column to be imported is the series name, the second column contains the category, and the third column contains the data.

- For scatter graphs, the first column to be imported is the series name, the second column is the data's x value, and the third column is the y value.

If a series or category already exists in the graph, the data is assigned to it. Otherwise, the series and categories are added to the graph.

You can add data to more than one series by specifying different series names in the first column.

Examples

This statement copies all data in the clipboard to the DataWindow dw_employee starting at the first column:

```
dw_employee.ImportClipboard( )
```

The following statement inserts data from the clipboard into the DataWindow dw_employee. It copies rows 2 through 30 and columns 3 through 8 on the clipboard to the DataWindow beginning in column 5. The result is 29 rows added to the DataWindow with data in columns 5 through 10:

```
dw_employee.ImportClipboard(2,30,3,8,5)
```

Syntax 2

If the clipboard contains the data shown below and the graph doesn't have any data yet, then the next statement produces a graph with two series and three categories. The clipboard data is:

```
Sales 94    Jan     3000
Sales 94    Mar     2200
Sales 94    May     2500
Sales 95    Jan     4000
Sales 95    Mar     3200
Sales 95    May     3500
```

This statement copies all the data in the clipboard, as shown above, to gr_employee:

```
gr_employee.ImportClipboard( )
```

This statement copies the data from the clipboard starting with row 2 column 3 and copying to row 30 column 5 to the graph gr_employee:

```
gr_employee.ImportClipboard(2, 30, 3)
```

See also

ImportFile
ImportString

ImportFile

Description

Inserts data into a DataWindow control or graph control from data in a file. The data can be tab-delimited text or dBase format 2 or 3. The format of the file depends on whether the target is a DataWindow or a graph and the type of graph. There are two syntaxes:

- To import rows into a DataWindow control, use Syntax 1.
- To add new series to a graph control, use Syntax 2.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

datawindowname.**ImportFile** (*filename* {, *startrow* {, *endrow* & {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Parameter	Description
datawindowname	The name of the DataWindow control to which you want to copy data from the specified file.
filename	A string whose value is the name of the file from which you want to copy data. The file must be an ASCII, tab-delimited file (.TXT) or a dBase format 2 or 3 file (.DBF). Specify the file's full name, which must end in the appropriate extension.
startrow (optional)	The number of the first row in the file that you want to copy. If the first row contains headings that you want to skip, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row in the file that you want to copy. The default is the rest of the rows.
startcolumn (optional)	The number of the first column in the file that you want to copy. The default is 1.
endcolumn (optional)	The number of the last column in the file that you want to copy. The default is the rest of the columns.
dwstartcolumn (optional)	The number of the first column in the DataWindow control that you want to receive data. The default is 1.

Return value 1

Long. Returns the number of rows that were imported if it succeeds and one of the following negative integers if an error occurs:

- *0* End of file; too many rows
- *-1* No rows
- *-2* Empty file
- *-3* Invalid argument
- *-4* Invalid input
- *-5* Could not open the file
- *-6* Could not close the file
- *-7* Error reading the text
- *-8* Not a TXT file
- *-9* The user canceled the import

Syntax 2

controlname.**ImportFile** (*filename* {, *startrow* {, *endrow* & {, *startcolumn* } } })

Parameter	Description
controlname	The name of the graph control to which you want to copy data from the specified file.
filename (optional)	A string containing the name of the file from which you want to copy data. The file must be an ASCII, tab-delimited file (.TXT) or a dBase format 2 or 3 file (.DBF). If you do not specify *filename*, ImportFile prompts the user for a file name.
startrow (optional)	The number of the first row in the file that you want to copy. If the first row contains headings that you want to skip, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row in the file that you want to copy. The default is the rest of the rows.
startcolumn (optional)	The number of the first column in the file that you want to copy. The default is 1.

Return value 2

Long. See Return value 1.

Usage

For a DataWindow control, the file should consist of rows of data. If the file includes column headings or row labels, set the *startrow* and *startcolumn* arguments to skip them. The data types and order of the DataWindow's columns must match the columns of data in the file.

The *startcolumn* and *endcolumn* arguments control the number of columns imported from the file and the number of columns in the DataWindow that are affected. The *dwstartcolumn* argument specifies the first DataWindow column to be affected. The following formula calculates the last DataWindow to be affected:

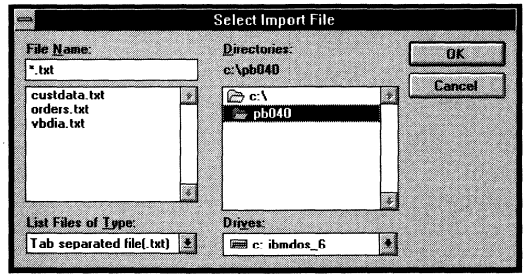
dwstartcolumn + (*endcolumn* - *startcolumn*)

For graph controls, ImportFile only uses three columns and ignores other columns. Each row of data must contain three pieces of information. The information depends on the type of graph:

- For all graph types except scatter, the first column to be imported is the series name, the second column contains the category, and the third column contains the data.
- For scatter graphs, the first column to be imported is the series name, the second column is the data's x value, and the third column is the y value.

You can add data to more than one series by specifying different series names in the first column.

To let users select the file to import, specify a null string for *filename*. PowerBuilder displays the Select Import File dialog.



Examples

This statement inserts all the data in the file D:\EMPLOYEE.TXT into dw_employee starting at the first column:

```
dw_employee.ImportFile("D:\EMPLOYEE.TXT")
```

This statement inserts the data from the file D:\EMPLOYEE.TXT into the DataWindow dw_employee. It copies rows 2 through 30 and columns 3 through 8 in the file to the DataWindow beginning in column 5. The result is 29 rows added to the DataWindow with data in columns 5 through 10:

```
dw_employee.ImportFile("D:\EMPLOYEE.TXT",  &
   2, 30, 3, 8, 5)
```

Syntax 2

This statement copies all the data in the file D:\EMPLOYEE.TXT to gr_employee starting at the first row:

```
gr_employee.ImportFile("D:\EMPLOYEE.TXT")
```

This statement copies the data from the file D:\EMPLOYEE.TXT starting with row 2 column 3 and ending with row 30 column 5 to the graph gr_employee:

```
gr_employee.ImportFile("D:\EMPLOYEE.TXT", 2, 30, 3)
```

The following example causes PowerBuilder to display the Specify Import File dialog:

```
string null_str
SetNull(null_str)
dw_main.ImportFile(null_str)
```

See also

ImportClipboard
ImportString

ImportString

Description

Inserts data points into a graph from tab-delimited data in a string. Inserts data into a DataWindow control from tab-delimited data in a string.

Inserts data into a DataWindow control or graph control from tab-delimited data in a string. The way data is arranged in the string into tab-delimited columns depends on whether the target is a DataWindow or a graph and the type of graph. There are two syntaxes:

- To import rows into a DataWindow control, use Syntax 1.
- To add new series to a graph control, use Syntax 2.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

datawindowname.**ImportString** (*string* {, *startrow* {, *endrow* & {, *startcolumn* {, *endcolumn* {, *dwstartcolumn* } } } } })

Parameter	Description
datawindowname	The name of the DataWindow control to which you want to copy data from the specified string.
string	A string from which you want to copy the data. The string should contain tab-delimited columns with one row per line (see Usage).
startrow (optional)	The number of the starting row of data in the string that you want to import. If the first row contains headings that you want to skip, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row of data in the string that you want to import. The default is a all the rows.
startcolumn (optional)	The number of the first column in the string that you want to import. The default is 1.
endcolumn (optional)	The number of the last column in the string that you want to import. The default is the rest of the columns.
dwstartcolumn (optional)	The number of the first column in the DataWindow control that you want to receive data. The default is 1.

Return value 1

Long. Returns the number of rows that were imported if it succeeds and one of the following negative integers if an error occurs:

- *-3* Invalid argument
- *-4* Invalid input

Syntax 2

controlname.**ImportString** (*string* {, *startrow* {, *endrow* & {, *startcolumn* }}})

Parameter	Description
controlname	The name of the graph control to which you want to copy data from the specified string.

Parameter	Description
string	A string from which you want to copy the data. The string's value should be tab-delimited columns with one data point per line (see Usage).
startrow (optional)	The number of the first row in the string that you want to copy. If the first row contains headings, set *startrow* to 2. The default is 1.
endrow (optional)	The number of the last row in the string that you want to copy. The default is the rest of the rows.
startcolumn (optional)	The number of the first series in the string that you want to import. The default is 1.

Return value 2

Long. Returns the number of data points that were imported if it succeeds and returns the following values if an error occurs:

- *0* End of file, too many rows
- -2 Not enough columns
- -3 Invalid argument
- -4 Invalid input

Usage

The format of the string is the same as if the data came from an ASCII file. The string must be formatted in tab-delimited columns and each line must end with a carriage return and a newline character (~r~n). If the string has four tab-delimited columns, one line might look like:

```
col1_data ~tcol2_data ~tcol3_data ~tcol4_data~r~n
```

For a DataWindow control, the string should consist of rows of data. If the data includes column headings or row labels, set the *startrow* and *startcolumn* arguments to skip them. The data types and order of the DataWindow's columns must match the columns of data in the string.

The *startcolumn* and *endcolumn* arguments control the number of columns imported from the string and the number of columns in the DataWindow that are affected. The *dwstartcolumn* argument specifies the first DataWindow column to be affected. The following formula calculates the last DataWindow to be affected:

dwstartcolumn + (*endcolumn* - *startcolumn*)

For graph controls, ImportString only uses three columns on each line and ignores other columns. The three columns must contain information that depends on the type of graph:

- For all graph types except scatter, the first column to be imported is the series name, the second column contains the category, and the third column contains the data.
- For scatter graphs, the first column to be imported is the series name, the second column is the data's x value, and the third column is the y value.

You can add data to more than one series by specifying different series names in the first column.

Examples

These statements copy all data in the string ls_Emp_Data to the DataWindow control dw_employee starting at the first column:

```
string ls_Emp_Data
ls_Emp_Data = . . .
dw_employee.ImportString(ls_Emp_Data)
```

The following statement stores data in the string ls_Text and imports it into the DataWindow dw_employee. The DataWindow is a report of department 100 and start and end dates of personnel. The string includes the department number and other information, which is not imported. ImportString imports rows 2 through 10 and columns 2 through 5 in the string to the DataWindow beginning in column 2. The result is 9 rows added to the DataWindow with data in columns 5 through 8:

```
string ls_text

ls_text =
"Dept~tLName~tFName~tStart~tEnd~tAmount~tOutcome
~r~n"
ls_text = ls_text +  &
   "100~tJones~tMary~tApr88~tJul94~t40~tG~r~n"
ls_text = ls_text +  &
   "100~tMarsh~tMarsha~tApr89~tJan92~t35~tG~r~n"
ls_text = ls_text +  &
   "100~tJames~tHarry~tAug88~tMar93~t22~tM~r~n"
. . .
ls_text = ls_text +  &
   "100~tWorth~tFrank~tSep87~tJun94~t55~tE~r~n"

dw_employee.ImportString(ls_text, 2, 10, 2, 5, 5)
```

Syntax 2

These statements copy the data from the string ls_Text starting with row 2 column 3 and ending with row 30 column 5 to the graph gr_employee:

```
string ls_Text
ls_Text = . . .
gr_employee.ImportString(ls_Text, 2, 30, 3)
```

The following script stores data for two series in the string ls_gr and imports the data into the graph gr_custbalance. The categories in the data are A, B, and C:

```
string ls_gr

ls_gr = "series1~tA~t12~r~n"
ls_gr = ls_gr + "series1~tB~t13~r~n"
ls_gr = ls_gr + "series1~tC~t14~r~n"
ls_gr = ls_gr + "series2~tA~t15~r~n"
ls_gr = ls_gr + "series2~tB~t14~r~n"
ls_gr = ls_gr + "series2~tC~t12.5~r~n"

gr_custbalance.ImportString(ls_gr, 1)
```

See also

ImportClipboard
ImportFile

InsertCategory

Description

Inserts a category on the category axis of a graph at the specified position. Existing categories are renumbered to keep the category numbering sequential.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**InsertCategory** (*categoryvalue*, *categorynumber*)

Parameter	Description
controlname	The name of the graph into which you want to insert a category.
categoryvalue	A value that is the category you want to insert. The category must be unique within the graph. The value you specify must be the same data type as the data type of the category axis.

Parameter	Description
categorynumber	The number of the category before which you want to insert the new category. To add the category at the end, specify 0. If the axis is sorted, the category will be integrated into the existing order, ignoring *categorynumber*.

Return value

Integer. Returns the number of the category if it succeeds and *-1* if an error occurs. If the category already exists, it returns the number of the existing category.

Usage

Categories are discrete "bins." Even on a date or time axis, each category is separate with no timeline-style connection between categories. Only scatter graphs, which do not have discrete categories, have a continuous category axis.

When the axis data type is string, category names are unique if they have different capitalization. Also, you can specify the empty string ("") as the category name. However, because category names must be unique, there can be only one category with that name.

When you use InsertCategory to create a new category, there will be holes in each of the series for that category. Use AddData or InsertData to create data points for the new category.

Equivalent syntax

If you want to add a category to the end of a series, you can use AddCategory instead, which requires fewer arguments.

This statement:

```
gr_data.InsertCategory("Qty", 0)
```

is equivalent to:

```
gr_data.AddCategory("Qty")
```

Example

These statements insert a category called Macs before the category named PCs in the graph gr_product_data:

```
integer CategoryNbr

// Get the number of the category.
CategoryNbr = FindCategory("PCs")
gr_product_data.InsertCategory("Macs", CategoryNbr)
```

In a graph reporting mail volume in the afternoon, these statements add three categories to a time axis:

```
catnum = gr_mail.InsertCategory(13:00, 0)
catnum = gr_mail.InsertCategory(12:00, 0)
catnum = gr_mail.InsertCategory(13:00, 0)
```

See also

AddData
AddCategory
FindCategory
FindSeries
InsertData
InsertSeries

InsertClass

Description

Inserts a new object of the specified OLE class in an OLE 2.0 control.

Syntax

ole2control.**InsertClass** (*classname*)

Parameter	Description
ole2control	The name of the OLE 2.0 control in which you want to create a new object
classname	A string whose value is the name of the class of the object you want to create

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Invalid class name
- *-9* Other error

Usage

Classnames are stored in the Registration database. Examples of classnames include:

- Excel.Sheet
- Excel.Chart
- Word.Document

Example

This example inserts an empty Excel spreadsheet into the OLE 2.0 control, ole_1:

```
integer result
result = ole_1.InsertClass("excel.sheet")
```

See also

InsertFile
InsertObject
LinkTo

InsertData

Description

Inserts a data point in a series of a graph. You can specify the category for the data point or its position in the series. Does not apply to scatter graphs.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**InsertData** (*seriesnumber*, *datapoint*, &
datavalue {, *categoryvalue* })

Parameter	Description
controlname	The name of the graph in which you want to insert data into a series.
seriesnumber	The number that identifies the series in which you want to insert data.
datapoint	The number of the data point before which you want to insert the data.
datavalue	The value of the data point you want to insert.
categoryvalue (optional)	The category for this data value on the category axis. The data type of *categoryvalue* should match the data type of the category axis. In most cases, you should include *categoryvalue*. Otherwise, an uncategorized value will be added to the series.

Return value

Integer. Returns the number of the data value if it succeeds and *-1* if an error occurs.

Usage

When you specify *datapoint* without specifying *categoryvalue*, InsertData inserts the data point in the category at that position, shifting existing data points to the following categories. The shift may cause there to be uncategorized data points at the end of the axis.

When you specify *categoryvalue*, InsertData ignores the position in *datapoint* and puts the data point in the specified category, replacing any data value that is already there. If the category does not exist, InsertData creates the category at the end of the axis.

To modify the value of a data point at a specified position, use ModifyData.

Scatter graphs
To add data to a scatter graph, use Syntax 2 of AddData.

Equivalent syntax

If you want to add a data point to the end of a series or to an existing category in a series, you can use AddData instead, which requires fewer arguments.

InsertData and ModifyData behave differently when you specify *datapoint* to indicate a position for inserting or modifying data. However, they behave the same as AddData when you specify a position of 0 and a category. All three modify the value of a data point when the category already exists. All three insert a category with a data value at the end of the axis when the category doesn't exist.

When you specify a position as well as a category, and that category already exists, InsertData ignores the position and modifies the data of the specified category, but ModifyData changes the category label at that position.

This statement:

```
gr_data.InsertData(1, 0, 44, "Qty")
```

is equivalent to:

```
gr_data.ModifyData(1, 0, 44, "Qty")
```

and is also equivalent to:

```
gr_data.AddData(1, 44, "Qty")
```

When you specify a position, the following statements are not equivalent.

InsertData ignores the position and modifies the data value of the Qty category:

```
gr_data.InsertData(1, 4, 44, "Qty")
```

while ModifyData changes the category label and the data value at position 4:

```
gr_data.ModifyData(1, 4, 44, "Qty")
```

Examples

Assuming the category label Jan does not already exist, these statements insert a data value in the series named Costs before the data point for Mar and assign the data point the category label Jan in the graph gr_product_data:

```
integer SeriesNbr, CategoryNbr

// Get the numbers of the series and category.
SeriesNbr = gr_product_data.FindSeries("Costs")
CategoryNbr = gr_product_data.FindCategory("Mar")
gr_product_data.InsertData(SeriesNbr,  &
   CategoryNbr, 1250, "Jan")
```

These statements insert the data value 1250 after the data value for Apr in the series named Revenues in the graph gr_product_data. The data is inserted in the category after Apr, and the rest of the data, if any, moves over a category:

```
integer SeriesNbr, CategoryNbr

// Get the number of the series and category.
CategoryNbr = gr_product_data.FindCategory("Apr")
SeriesNbr = gr_product_data.FindSeries("Revenues")

gr_product_data.InsertData(SeriesNbr,  &
   CategoryNbr + 1, 1250)
```

See also

AddData
FindCategory
FindSeries
GetData

InsertFile

Description

Inserts an object into an OLE 2.0 control. A copy of the specified file is embedded in the OLE object.

Syntax

ole2control.**InsertFile** (*filename*)

Parameter	Description
ole2control	The name of the OLE 2.0 control.
filename	A string whose value is the name of the file whose contents you want to be the data in the embedded OLE object. *Filename* should include the file's path.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* File not found
- *-9* Other error

Usage

The contents of the specified file is embedded in the OLE object. There is no further link between the object in PowerBuilder and the file.

Example

This example creates a new OLE object in the control ole_1. It is an Excel object and contains data from the spreadsheet EXPENSE.XLS:

```
integer result
result = ole_1.InsertFile("c:\xls\expense.xls")
```

See also

InsertClass
InsertObject
LinkTo
Paste

InsertItem

Description

Inserts an item into the list of values in a listbox.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**InsertItem** (*item*, *index*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox into which you want to insert an item
item	A string whose value is the text of the item you want to insert
index	The number of the item in the list before which you want to insert the item

Return value

Integer. Returns the final position of the item. Returns *-1* if an error occurs.

Usage

InsertItem inserts the new item before the item identified by *index*. If the items in *listboxname* are sorted (its Sorted attribute is TRUE), PowerBuilder resorts the items after the new item is inserted. The return value reflects the new item's final position in the list.

> **VBX controls**
> If you have created a VBX user object using a VBX control that supports the AddItem method, use the AddItem or InsertItem function call instead of the AddItem method.

Examples

This statement inserts the item Run Application before the fifth item in lb_actions:

```
lb_actions.InsertItem("Run Application", 5)
```

If the Sorted attribute is FALSE, the statement above returns 5 (the previous item 5 becomes item 6). If the Sorted attribute is TRUE, the list is sorted after the item is inserted and the function returns the index of the final position of the item.

If the ListBox lb_Cities has the following items in its list and its Sorted attribute is set to TRUE, then the following example inserts Denver at the top, sorts the list, and sets li_pos to 4. If the ListBox's Sorted attribute is FALSE, then the statement inserts Denver at the top of the list and sets li_pos to 1. The list is:

Albany
Boston
Chicago
New York

The example code is:

```
string ls_City = "Denver"
integer li_pos
li_pos = lb_Cities.InsertItem(ls_City, 1)
```

See also

AddItem
DeleteItem
FindItem
Reset
TotalItems

InsertObject

Description

Displays the standard Insert Object dialog, allowing the user to choose a new or existing OLE object, and inserts the selected object in the OLE 2.0 control.

Syntax

ole2control.**InsertObject** ( )

Parameter	Description
ole2control	The name of the OLE 2.0 control in which you want to insert an object

Return value

Integer. Returns *0* if it succeeds and one of the following values if an error occurs:

- *1* User canceled out of dialog
- *-9* Other error

Example

This example displays the standard Insert Object so that the user can select an OLE object. InsertObject inserts the selected object in the ole_1 control:

```
integer result
result = ole_1.InsertObject( )
```

See also

InsertClass
InsertFile
LinkTo

InsertRow

Description

Inserts a row in a DataWindow. If any columns have default values, the row is initialized with these values before it is displayed.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**InsertRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to insert a row
row	A long identifying the row before which you want to insert the row. To insert a row at the end, specify 0.

Return value

Long. Returns the number of the row that was added if it succeeds and *-1* if an error occurs.

Usage

InsertRow simply inserts the row without changing the display or the current row. To scroll to the row and make it the current row, call ScrollToRow. To simply make it the current row, call SetRow.

Examples

This statement inserts an initialized row before row 7 in dw_Employee:

```
dw_Employee.InsertRow(7)
```

This example inserts an initialized row after the last row in dw_employee, then scrolls to the row, which makes it current:

```
long ll_newrow
ll_newrow = dw_employee.InsertRow(0)
dw_employee.ScrollToRow(ll_newrow)
```

See also

DeleteRow
Update

InsertSeries

Description

Inserts a series in a graph at the specified position. Existing series in the graph are renumbered to keep the numbering sequential.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax

controlname.**InsertSeries** (*seriesname*, *seriesnumber*)

Parameter	Description
controlname	The name of the graph in which you want to insert a series.
seriesname	A string containing the name of the series you want to insert. The series name must be unique within the graph.
seriesnumber	The number of the series before which you want to insert the new series. To add the new series at the end, specify 0.

Return value

Integer. Returns the number of the series if it succeeds and *-1* if an error occurs. If the series named in *seriesname* exists already, it returns the number of the existing series.

Usage

Series names are unique if they have different capitalization.

Equivalent syntax

If you want to add a series to the end of the list, you can use AddSeries instead, which requires fewer arguments.

This statement:

```
gr_data.InsertSeries("Costs", 0)
```

is equivalent to:

```
gr_data.AddSeries("Costs")
```

Examples

These statements insert a series before the series named Income in the graph gr_product_data:

```
integer SeriesNbr

// Get the number of the series.
SeriesNbr = FindSeries("Income")
gr_product_data.InsertSeries("Costs", SeriesNbr)
```

See also

AddData
AddSeries
FindCategory
FindSeries
InsertCategory
InsertData

Int

Description

Determines the largest whole number less than or equal to a number.

Syntax

Int (*n*)

Parameter	Description
n	The number for which you want the largest whole number that is less than or equal to it

Return value

Integer. Returns the largest whole number less than or equal to *n*.

Examples

These statements return 3.0:

```
Int(3.2)
Int(3.8)
```

The following statements return -4.0:

```
Int(-3.2)
Int(-3.8)
```

These statements remove the decimal portion of the variable and store the resulting integer in li_nbr:

```
integer li_nbr
li_nbr = Int(3.2)     // li_nbr = 3
```

See also

Ceiling
Round
Truncate
Int in Chapter 2, "DataWindow Painter Functions"

Integer

Description

Converts the value of a string to an integer or obtains an integer value that is stored in a blob.

Syntax

Integer (*stringorblob*)

Parameter	Description
stringorblob	The string whose value you want returned as an integer or a blob in which the first value is the integer value. The rest of the contents of the blob is ignored.

Return value

Integer. Returns the value of *stringorblob* as an integer if it succeeds and *0* if *stringorblob* is not a number.

Usage

To distinguish between a string whose value is the number 0 and a string whose value is not a number, use the IsNumber function before calling the Integer function.

Examples

This statement returns the string 24 as an integer:

```
Integer("24")
```

This statement returns the contents of the SingleLineEdit sle_Age as an integer:

```
Integer(sle_Age.Text)
```

This statement returns 0:

```
Integer("3ABC")  // 3ABC is not a number.
```

This example checks whether the text of sle_data is a number before converting, which is necessary if the user might legitimately enter 0:

```
integer li_new_data
IF IsNumber(sle_data.Text) THEN
   li_new_data = Integer(sle_data.Text)
ELSE
   SetNull(li_new_data)
END IF
```

After assigning blob data from the database to lb_blob, the following example obtains the integer value stored at position 20 in the blob:

```
integer i
i = Integer(BlobMid(lb_blob, 20, 2))
```

See also

Double
Dec
IsNumber
Long
Real
Integer in Chapter 2, "DataWindow Painter Functions"

IntHigh

Description

Returns the high word of a long value.

Syntax

IntHigh (*long*)

Parameter	Description
long	A long value

Return value

Integer. Returns the high word of *long* if it succeeds and *-1* if an error occurs.

Usage

One use for IntHigh is for decoding values returned by external C functions and Windows messages.

Example

These statements decode a long value LValue into its low and high integers:

```
integer   nLow, nHigh
long  LValue = 274489
nLow = IntLow (LValue)  //The Low Integer is 12345.
nHigh = IntHigh(LValue) //The High Integer is 4.
```

See also

IntLow

IntLow

Description

Returns the low word of a long value.

Syntax

IntLow (*long*)

Parameter	Description
long	A long value

Return value

Integer. Returns the low word of *long* if it succeeds and *-1* if an error occurs.

Usage

One use for IntLow is for decoding values returned by external C functions and Windows messages.

Example

These statements decode a long value LValue into its low and high integers:

```
integer  nLow, nHigh
long  LValue = 12345
nLow = IntLow (LValue)  //The Low Integer is 12345.
nHigh = IntHigh(LValue) //The High Integer is 0.
```

See also

IntHigh

IsDate

Description

Tests whether a string value is a valid date.

Syntax

IsDate (*datevalue*)

Parameter	Description
datevalue	A string whose value you want to test to determine whether it is a valid date

Return value

Boolean. Returns TRUE if *datevalue* is a valid date and FALSE if it is not.

Usage

You can use IsDate to test whether a user-entered date is valid before you convert it to a date data type. To convert a value into a date value, use the Date function.

Examples

This statement returns TRUE:

```
IsDate("Jan 1, 95")
```

This statement returns FALSE:

```
IsDate("Jan 32, 1997")
```

If the SingleLineEdit sle_Date_Of_Hire contains 7/1/91, these statements store 1991-07-01 in HireDate:

```
Date HireDate
IF IsDate(sle_Date_Of_Hire.text) THEN
   HireDate = Date(sle_Date_Of_Hire.text)
END IF
```

See also IsDate in Chapter 2, "DataWindow Painter Functions"

IsNull

Description Reports whether the value of a variable or expression is NULL.

Syntax **IsNull** (*any*)

Parameter	Description
any	A variable or expression that you want to test to determine whether its value is NULL

Return value Boolean. Returns TRUE if *any* is NULL and FALSE if it is not.

Usage Use IsNull to test whether a user-entered value or a value retrieved from the database is NULL. IsNull works for all data types but does not work for arrays.

If one or more columns in a DataWindow are required columns, that is, they must contain data, you don't want to update the database if the columns have NULL values. You can use FindRequired to find rows in which those columns have NULL values, instead of using IsNull to evaluate each row and column.

> **Tip**
> To set a variable to NULL, use the SetNull function.

Example These statements set lb_test to TRUE:

```
integer a, b
boolean lb_test
SetNull(b)
lb_test = IsNull(a + b)
```

See also

SetNull
IsNull in Chapter 2, "DataWindow Painter Functions"

IsNumber

Description

Reports whether the value of a string is a number.

Syntax

IsNumber (*string*)

Parameter	Description
string	A string whose value you want to test to determine whether it is a valid PowerScript number

Return value

Boolean. Returns TRUE if *string* is a valid PowerScript number and FALSE if it is not.

Usage

Use IsNumber to check that text in an edit control can be converted to a number.

To convert a string to a specific numeric data type, use the Double, Dec, Integer, Long, or Real function.

Examples

This statement returns TRUE:

```
IsNumber("32.65")
```

This statement returns FALSE:

```
IsNumber("A16")
```

If the SingleLineEdit sle_Age contains 32, these statements store 32 in li_YearsOld:

```
integer li_YearsOld
IF IsNumber(sle_Age.Text) THEN
   li_YearsOld = Integer(sle_Age.Text)
END IF
```

See also

Double
Dec

Integer
Long
Real
IsNumber in Chapter 2, "DataWindow Painter Functions"

IsSelected

Description

Determines whether a row is selected in a DataWindow. A selected row is highlighted using reverse video.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**IsSelected** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that contains the row you want to test
row	A long identifying the row you want to test to see if it is selected

Return value

Boolean. Returns TRUE if *row* in *datawindowname* is selected and FALSE if it is not selected. If *row* is greater than the number of rows in *datawindowname* or is 0 or negative, IsSelected also returns FALSE.

Usage

You can call IsSelected in a script for the Clicked event to determine whether the row the user clicked was selected.

Example

This code calls IsSelected to test whether the current row in dw_employee is selected. If the row is selected, SelectRow deselects it; if it is not selected, SelectRow selects it:

```
integer CurRow
boolean result

CurRow = dw_employee.GetRow()
result = dw_employee.IsSelected(CurRow)
```

```
IF result THEN
   dw_employee.SelectRow(CurRow, FALSE)
ELSE
   dw_employee.SelectRow(CurRow, TRUE)
END IF
```

The following code uses the NOT operator on the return value of IsSelected to accomplish the same result as the IF/THEN/ELSE statement above:

```
integer CurRow
boolean result

CurRow = dw_employee.GetRow()
dw_employee.SelectRow(CurRow,  &
      NOT dw_employee.IsSelected(CurRow))
```

See also

SelectRow
IsSelected in Chapter 2, "DataWindow Painter Functions"

IsTime

Description

Reports whether the value of a string is a valid time value.

Syntax

IsTime (*timevalue*)

Parameter	Description
timevalue	A string whose value you want to test to determine whether it is a valid time

Return value

Boolean. Returns TRUE if *timevalue* is a valid time and FALSE if it is not.

Usage

Use IsTime to test to whether a value a user enters in an edit control is a valid time.

To convert a string to an time value, use the Time function.

Examples

This statement returns TRUE:

```
IsTime("8:00:00 am")
```

This statement returns FALSE:

```
IsTime("25:00")
```

If the SingleLineEdit sle_EndTime contains 4:15 these statements store 04:15:00 in lt_QuitTime:

```
Time lt_QuitTime
IF IsTime(sle_EndTime.Text) THEN
   lt_QuitTime = Time(sle_EndTime.Text)
END IF
```

See also

Time
IsTime in Chapter 2, "DataWindow Painter Functions"

IsValid

Description

Determines whether an object variable is instantiated, that is, whether its value is a valid object handle.

Syntax

IsValid (*objectname*)

Parameter	Description
objectname	The name of an object

Return value

Boolean. Returns a boolean value indicating whether *objectname* has been created. If objectname is NULL, IsValid returns NULL.

Usage

Use IsValid instead of the Handle function to determine whether a window is open.

Example

This statement determines whether the window w_emp is open and if it is not, opens it:

```
IF IsValid(w_emp) = FALSE THEN Open(w_emp)
```

See also

Handle

KeyDown

Description

Determines whether the user pressed the specified key on the computer keyboard.

Syntax

KeyDown (*keycode*)

Parameter	Description
keycode	A value of the KeyCode enumerated data type that identifies a key on the computer keyboard or an integer whose value is the ASCII code for a key. A table of KeyCode values follows Examples.

Return value

Boolean. Returns TRUE if *keycode* was pressed and FALSE if it was not.

Usage

Call KeyDown in the Key event to determine whether the user pressed a particular key. The Key event occurs whenever the user presses a key as long as the insertion point is not in a line edit.

A table of KeyCode values follows the examples. For KeyCode names that aren't obvious, a comment identifies the KeyCode's key.

Examples

The following code checks whether the user pressed F1 key or the CTRL key and executes some statements appropriate to the key pressed:

```
IF KeyDown(KeyF1!) THEN
   . . .  // Statements for the F1 key
ELSEIF KeyDown(KeyControl!) THEN
   . . .  // Statements for the CTRL key
END IF
```

This statement tests whether the user pressed capital A. The ASCII value for A is 65:

```
IF KeyDown(65) THEN ...
```

KeyCode Values

keyBack! — Backspace
keyTab!
keyEnter!
keyShift!
keyControl!
keyAlt!
keyPause!
keyCapsLock!
keyEscape!
keySpaceBar!
keyPageUp!
keyPageDown!
keyEnd!
keyHome!
keyLeftArrow!
keyUpArrow!
keyRightArrow!
keyDownArrow!
keyPrintScreen!
keyInsert!
keyDelete!

key0! — 0 and)
key1! — 1 and !
key2! — 2 and @
key3! — 3 and #
key4! — 4 and $
key5! — 5 and %
key6! — 6 and ^
key7! — 7 and &
key8! — 8 and *
key9! — 9 and (
keyA!
keyB!
keyC!
keyD!
keyE!
keyF!
keyG!
keyH!
keyI!
keyJ!
keyK!
keyL!
keyM!
keyN!
keyO!
keyP!

keyQ!
keyR!
keyS!
keyT!
keyU!
keyV!
keyW!
keyX!
keyY!
keyZ!
keyNumpad0!
keyNumpad1!
keyNumpad2!
keyNumpad3!
keyNumpad4!
keyNumpad5!
keyNumpad6!
keyNumpad7!
keyNumpad8!
keyNumpad9!
keyF1!
keyF2!
keyF3!
keyF4!
keyF5!

keyF6!
keyF7!
keyF8!
keyF9!
keyF10!
keyF11!
keyF12!
keyMultiply! — * on numeric keypad
keyAdd! — + on numeric keypad
KeySubtract! — - on numeric keypad
keyDecimal! — . on numeric keypad when NumLock is on
keyDivide! — / on numeric keypad
keyNumLock!
keyScrollLock!
keyQuote! — ' and "
keyEqual! — = and +
keyComma! — , and <
keyDash! — - and _
keyPeriod! — . and >
keySlash! — / and ?
keyBackQuote! — ` and ~
keyLeftBracket! — [and {
keyBackSlash! — \ and |
keyRightBracket! —] and }
keySemiColon! — ; and :

Left

Description

Obtains a specified number of characters from the beginning of a string.

Syntax

Left (*string, n*)

Parameter	Description
string	The string containing the characters you want
n	A long specifying the number of characters you want

Return value

String. Returns the leftmost *n* characters in *string* if it succeeds and the empty string ("") if an error occurs.

If *n* is greater than or equal to the length of the string, Left returns the entire string. It does not add spaces to make the return value's length equal to *n*.

Examples

This statement returns BABE:

```
Left("BABE RUTH", 4)
```

This statement returns BABE RUTH:

```
Left("BABE RUTH", 40)
```

These statements store the first 40 characters of the text in the SingleLineEdit sle_address in emp_address:

```
string emp_address
emp_address = Left(sle_address.Text, 40)
```

For sample code that uses Left to parse two tab-separated values, see the Pos function.

See also

Mid
Pos
Right
Left in Chapter 2, "DataWindow Painter Functions"

LeftTrim

Description

Removes spaces from the beginning of a string.

Syntax

LeftTrim (*string*)

Parameter	Description
string	The string you want returned with leading spaces deleted

Return value

String. Returns a copy of *string* with leading spaces deleted if it succeeds and the empty string ("") if an error occurs.

Examples

This statement returns RUTH:

```
LeftTrim("    RUTH")
```

These statements delete leading spaces from the text in the MultiLineEdit mle_name and store the result in emp_name:

```
string emp_name
emp_name = LeftTrim(mle_name.Text)
```

See also

RightTrim
Trim
LeftTrim in Chapter 2, "DataWindow Painter Functions"

Len

Description

Reports the length of a string or a blob.

Syntax

Len (*stringorblob*)

Parameter	Description
stringorblob	The string or blob for which you want the length

Return value

Long. Returns a long whose value is the length of *stringorblob* if it succeeds and *-1* if an error occurs.

Usage

Len counts the number of characters in a string. The NULL that terminates a string is not included in the count.

If you specify a size when you declare a blob, that is the size reported by Len. If you don't specify a size for the blob, PowerBuilder assigns it a size the first time you assign data to the blob, which becomes the size reported by Len. Initially, Len reports that the blob's length is 0.

Examples

This statement returns 0:

```
Len("")
```

These statements store in the variable s_address_len the length of the text in the SingleLineEdit sle_address:

```
long s_address_len
s_address_len = Len(sle_address.Text)
```

The following scenarios illustrate how the declaration of blobs affects their length, as reported by Len.

In the first example, an instance variable called ib_blob is declared but not initialized with a size. If you call Len before data is assigned to ib_blob, Len returns 0. After data is assigned, Len returns the blob's new length.

The declaration of the instance variable is:

```
blob ib_blob
```

The sample code is:

```
long ll_len
ll_len = Len(ib_blob)      // ll_len set to 0
ib_blob = Blob( "Test String")
ll_len = Len(ib_blob)      // ll_len set to 11
```

In the second example, ib_blob is initialized to the size 100 when it is declared. When you call Len for ib_blob, it always returns 100. This example uses BlobEdit, instead of Blob, to assign data to the blob because its size is already established. The declaration of the instance variable is:

```
blob{100} ib_blob
```

The sample code is:

```
long ll_len
ll_len = Len(ib_blob)      // ll_len set to 100
BlobEdit(ib_blob, 1, "Test String")
ll_len = Len(ib_blob)      // ll_len set to 100
```

See also

Len in Chapter 2, "DataWindow Painter Functions"

Length

Description

Reports the length in bytes of an open OLE stream.

Applies to

oleStream objects

Syntax

olestream.**Length** (*sizevar*)

Parameter	Description
olestream	The name of an OLE stream variable that has been opened
sizevar	A long variable in which Length will store the size of *olestream*

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Stream is not open
- *-9* Other error

Example

This example opens an OLE object in the file MYSTUFF.OLE and assigns it to the oleStorage object stg_stuff. Then it opens the stream called info in stg_stuff and assigns it to the stream object olestr_info. Finally, it finds out the stream's length and stores the value in the variable info_len.

The example doesn't check the functions' return values for success, but you should be sure to check the return values in your code:

```
boolean lb_memexists
oleStorage stg_stuff
oleStream olestr_info
long info_len

stg_stuff = CREATE oleStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info",  &
   stgRead!, stgExclusive!)
olestr_info.Length(info_len)
```

See also

Open
Read
Seek
Write

LibraryCreate

Description

Creates an empty PowerBuilder library with optional comments.

Syntax

LibraryCreate (*libraryname* {, *comments* })

Parameter	Description
libraryname	A string whose value is the name of the PowerBuilder library you want to create. If you want to create the library somewhere other than the current directory, enter the full path name.
comments (optional)	A string whose value is the comments you want to associate with the library.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

LibraryCreate creates a PowerBuilder library file (PBL) in the current directory, unless you specify a directory path as part of *libraryname*. If you don't specify an extension, LibraryCreate adds the extension .PBL.

Example

This statement in Windows 3.1 or Windows NT creates a library named dwTemp in the pb4 directory on drive C and associates a comment with the library:

```
LibraryCreate("c:\pb4\dwTemp.pbl",  &
"Temporary library for dynamic DataWindows")
```

This statement creates the same library in the PowerBuilder 4 folder on a Macintosh drive:

```
LibraryCreate("HardDisk:PowerBuilder 4:dwTemp.pbl",&
"Temporary library for dynamic DataWindows")
```

See also

LibraryDelete
LibraryDirectory
LibraryExport
LibraryImport

LibraryDelete

Description

Deletes a library file or, if you specify a DataWindow object, deletes the DataWindow object from the library.

Syntax

LibraryDelete (*libraryname* {, *objectname*, *objecttype* })

Parameter	Description
libraryname	A string whose value is the name of the PowerBuilder library you want to delete or from which you want to delete a DataWindow object. If you do not specify a full path, LibraryDelete uses the system's standard file search order to find the file.
objectname (optional)	A string whose value is the name of the DataWindow object you want to delete from *libraryname*.
objecttype (optional)	A value of the LibImportType enumerated data type identifying the type of object you want to delete. The only supported object type is ImportDataWindow!.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

You can delete DataWindow objects from a library in a script with the LibraryDelete function. To delete other types of objects, use the Library painter.

Examples

This statement deletes a library called dwTemp in the current directory and on the current application library path:

```
LibraryDelete("dwTemp.pbl")
```

This statement deletes the DataWindow object d_emp from the library called dwTemp. The value for the *libraryname* argument is appropriate for Windows 3.1 or Windows NT. For Macintosh, substitute an appropriate path:

```
LibraryDelete("c:\pb4\dwTemp.pbl", "d_emp", &
   ImportDataWindow!)
```

See also

LibraryCreate
LibraryDirectory
LibraryExport
LibraryImport

LibraryDirectory

Description

Obtains a list of the objects in a PowerBuilder library. The information provided is the object name, the date and time it was last modified, and any comments for the object. You can get a list of all objects or just objects of a specified type.

Syntax

LibraryDirectory (*libraryname, objecttype*)

Parameter	Description
libraryname	A string whose value is the name of the PowerBuilder library for which you want the contents. If you do not specify a full path, LibraryDirectory uses the operating system's standard file search order to find the file.

Parameter	Description
objecttype	A value of the LibDirType enumerated data type identifying the type of objects you want listed: ♦ DirAll! — All objects ♦ DirApplication! — Application objects ♦ DirDataWindow! — DataWindow objects ♦ DirFunction! — Function objects ♦ DirMenu! — Menu objects ♦ DirPipeline! — Pipeline objects ♦ DirProject! — Project objects ♦ DirQuery! — Query objects ♦ DirStructure! — Structure objects ♦ DirUserObject! — User objects ♦ DirWindow! — Window objects

Return value

String. LibraryDirectory returns a tab-separated list with one object per line. The format of the list is:

```
name ~t date/time modified ~t comments ~n
```

Returns the empty string ("") if an error occurs.

Usage

You can display the result of LibraryDirectory in a DataWindow control by passing the returned string to the ImportString function for that DataWindow. The DataWindow that contains three string columns. The columns must be wide enough to fit the data in the input string. If not, PowerBuilder will report validation errors.

For an example of parsing tab-delimited data, see the Pos function.

Example

This code imports the string returned by LibraryDirectory to the DataWindow dw_list and then redraws the dw_list. The DataWindow was defined with an external source and three string columns. The value for the *libraryname* argument is appropriate for Windows 3.1 or Windows NT. For Macintosh, substitute an appropriate path:

```
String ls_entries

ls_entries = LibraryDirectory(  &
   "c:\pb4\dwTemp.pbl", DirUserObject!)
dw_list.SetRedraw(FALSE)
dw_list.Reset( )
dw_list.ImportString(ls_Entries)
dw_list.SetRedraw(TRUE)
```

See also

ImportString
LibraryCreate
LibraryDelete
LibraryExport
LibraryImport

LibraryExport

Description

Exports an object from a library. The object is exported as syntax.

Syntax

LibraryExport (*libraryname, objectname, objecttype*)

Parameter	Description
libraryname	A string whose value is the name of the PowerBuilder library from which you want to export an object. If you do not specify a full path, LibraryExport uses the system's standard file search order to find the file.
objectname	A string whose value is the name of the object you want to export.

Parameter	Description
objecttype	A value of the LibExportType enumerated data type identifying the type of objects you want to export: ♦ ExportApplication! — Application object ♦ ExportDataWindow! — DataWindow object ♦ ExportFunction! — Function object ♦ ExportMenu! — Menu object ♦ ExportPipeline! — Pipeline objects ♦ ExportProject! — Project objects ♦ ExportQuery! — Query objects ♦ ExportStructure! — Structure object ♦ ExportUserObject! — User objects ♦ ExportWindow! — Window object

Return value

String. Returns the syntax of the object if it succeeds. The syntax is the same as the syntax returned when you export an object in the Library painter except that LibraryExport does not include an export header. Returns the empty string ("") if an error occurs.

Example

These statements export the DataWindow object dw_emp from the library called dwTemp to a string named ls_dwsyn and then use it to create a DataWindow. The value for the *libraryname* argument is appropriate for Windows 3.1 or Windows NT. For Macintosh, substitute an appropriate path:

```
String ls_dwsyn, ls_errors
ls_dwsyn = LibraryExport("c:\pb4\dwTemp.pbl", &
   "d_emp", ExportDataWindow!)
dw_1.Create(ls_dwsyn, ls_errors)
```

See also

Create
LibraryCreate
LibraryDelete
LibraryDirectory
LibraryImport

LibraryImport

Description

Imports a DataWindow object into a library. LibraryImport uses the syntax of the DataWindow object, which is specified in text format, to recreate the object in the library.

Syntax

LibraryImport (*libraryname*, *objectname*, *objecttype*, & *syntax*, *errors* {,*comments*})

Parameter	Description
libraryname	A string specifying the name of the PowerBuilder library into which you want to import the entry. If you do not specify a full path, LibraryImport uses the system's standard file search order to find the file.
objectname	A string specifying the name of the DataWindow object you want to import.
objecttype	A value of the LibImportType enumerated data type identifying the type of object you want to delete. The only supported object type is ImportDataWindow!.
syntax	A string specifying the syntax of the DataWindow object you want to import.
error	A string variable that you want to fill with any error messages that occur.
comments (optional)	A string specifying the comments you want to associate with the entry.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

When you import a DataWindow, any errors that occur are stored in the string variable you specify for the *error* argument.

When your application creates a DataWindow dynamically during execution, you can use LibraryImport to save that DataWindow object in a library.

Examples

These statements import the DataWindow object d_emp into the library called dwTemp and store any errors in ErrorBuffer. Note that the syntax is obtained by using the Describe function:

```
string dwsyntax, ErrorBuffer
integer rtncode

dwsyntax = dw_1.Describe("DataWindow.Syntax")
rtncode = LibraryImport("c:\pb4\dwTemp.pbl",  &
   "d_emp", ImportDataWindow!,  &
   dwsyntax, ErrorBuffer )
```

These statements import the DataWindow object d_emp into the library called dwTemp, store any errors in ErrorBuffer, and associate the comment Employee DataWindow 1 with the entry:

```
string dwsyntax, ErrorBuffer
integer rtncode

dwsyntax = dw_1.Describe("DataWindow.Syntax")
rtncode = LibraryImport("c:\pb4\dwTemp.pbl",  &
   "d_emp", ImportDataWindow!,  &
   dwsyntax, ErrorBuffer,  &
   "Employee DataWindow 1")
```

See also

Describe
LibraryCreate
LibraryDelete
LibraryDirectory
LibraryExport

LineCount

Description

Determines the number of lines in an edit control that allows multiple lines.

Applies to

MultiLineEdit, EditMask, and DataWindow controls

Syntax

editname.**LineCount** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask. or MultiLineEdit for which you want the number of lines

Return value

Integer. Returns the number of lines in *editname* if it succeeds and *-1* if an error occurs.

Usage

When you call LineCount for a DataWindow, it reports the number of lines in the edit control over the current row and column. A user can enter multiple lines in a DataWindow column only if it has a text data type and its box is large enough to display those lines. The size of the column's box determines the number of lines allowed in the column. When the user is typing, lines do not wrap automatically; the user must press ENTER to type additional lines.

In a MultiLineEdit control, lines wrap when the user's typing fills the control horizontally, unless either the HScrollBar or AutoHScroll attribute is TRUE. If horizontal scrolling is enabled with these attributes, the user must press ENTER to type additional lines.

LineCount counts each visible line, whether it was the result of wrapping or carriage returns.

Examples

If the MultiLineEdit mle_Instructions has 9 lines, the following example sets li_Count to 9:

```
integer li_Count
li_Count = mle_Instructions.LineCount( )
```

These statements display a MessageBox if fewer than two lines have been entered in the MultiLineEdit mle_Address:

```
integer li_Lines
li_Lines = mle_Address.LineCount( )
IF li_Lines < 2 THEN
   MessageBox("Warning", "2 lines are required.")
END IF
```

LineLength

Description

Determines the length of the line containing the insertion point in a MultiLineEdit or EditMask control.

Applies to

MultiLineEdit and EditMask controls

Syntax

editname.**LineLength** ( )

Parameter	Description
editname	The name of the MultiLineEdit or EditMask containing the text in which you want to determine the length of the line containing the insertion point

Return value

Integer. Returns the length of the line containing the insertion point in *editname*. Returns *-1* if an error occurs.

Usage

PowerBuilder remembers where the insertion point is in each editable control. When the user moves the focus to another control, you can still find out the length of the line most recently edited by calling the LineLength function for that control.

> **Insertion point in editable controls**
> Because PowerBuilder remembers the position of the insertion point, users can resume editing at the insertion point if they make the control active by tabbing to it. When users make a control active by clicking on it, they move the insertion point as well.

For an EditMask control, LineLength reports the length of the mask, regardless of the number of characters the user has entered.

Example

If the insertion point is positioned anywhere in line 5 of mle_Contact and line 5 contains the text Select All, il_linelength is set to 10 (the length of line 5):

```
integer li_linelength
li_linelength = mle_Contact.LineLength( )
```

See also

Position
SelectedLine
SelectedStart
TextLine

LinkTo

Description

Establishes a link between an OLE 2.0 control and a file or an item within the file.

Syntax

ole2control.**LinkTo** (*filename* {, *sourceitem* })

Parameter	Description
ole2control	The name of the OLE 2.0 control in which you want to insert a linked object.
filename	A string whose value is the filename containing the data that you want to insert in *ole2control*, with a link connecting the object in PowerBuilder to the original data. If you don't specify *sourceitem*, a link is established with the whole file.
sourceitem (optional)	A string that names an item within *filename* to which you want to link. The way you specify *sourceitem* is determined by the OLE server application.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* File not found
- *-2* Item not found
- *-9* Other error

Example

This example creates an object in the OLE 2.0 control, ole_1. The object is linked to the file C:\XLS\EXPENSE.XLS:

```
integer result
result = ole_1.LinkTo("c:\xls\expense.xls")
```

This example links to a section of rows and columns in the same spreadsheet as in the previous example:

```
integer result
result = ole_1.LinkTo("c:\xls\expense.xls",  &
   "R1C1:R5C5")
```

See also

InsertFile
InsertObject

PasteLink
PasteSpecial

Log

Description

Determines the natural logarithm of a number.

Syntax

Log (*n*)

Parameter	Description
n	The number for which you want the natural logarithm (base e). The value of *n* must be greater than 0.

Return value

Double. Returns the natural logarithm of *n*. An execution error occurs if *n* is negative or zero.

> **Tip**
> The inverse of the Log function is the Exp function.

Examples

This statement returns 2.302585092:

```
Log(10)
```

This statement returns –.693147 . . . :

```
Log(0.5)
```

Both these statements result in an error during execution:

```
Log(0)
```

```
Log(-2)
```

After the following statements execute, the value of a is 200:

```
double a, b = Log(200)
a = Exp(b)  // a = 200
```

See also

Exp
LogTen
Log in Chapter 2, "DataWindow Painter Functions"

LogTen

Description

Determines the base 10 logarithm of a number.

Syntax

LogTen (*n*)

Parameter	Description
n	The number for which you want the base 10 logarithm. The value of *n* must not be negative.

Return value

Double. Returns the base 10 logarithm of *n*. An execution error occurs if *n* is negative.

> **Tip**
> The expression 10^*n* is the inverse for LogTen(*n*). To obtain the value of *n* in the equation r = LogTen(*n*), use *n* = 10^r.

Examples

This statement returns 1:

```
LogTen(10)
```

The following statements both return 0:

```
LogTen(1)
LogTen(0)
```

This statement results in an execution error:

```
LogTen(-2)
```

After the following statements execute, the value of a is 200:

```
double a, b = LogTen(200)
a = 10^b    // a = 200
```

See also

Exp
Log
LogTen in Chapter 2, "DataWindow Painter Functions"

Long

Description

Converts data into data of type long. There are two syntaxes:

- To combine two unsigned integers into a long value, use Syntax 1.
- To convert a string whose value is a number into a long or to obtain a long value stored in a blob, use Syntax 2.

Syntax 1

Long (*lowword, highword*)

Parameter	Description
lowword	An UnsignedInteger to be the low word in the long
highword	An UnsignedInteger to be the high word in the long

Return value 1

Long. Returns the long if it succeeds and *-1* if an error occurs.

Syntax 2

Long (*stringorblob*)

Parameter	Description
stringorblob	The string you want returned as a long or a blob in which the first value is the long value. The rest of the contents of the blob is ignored.

Return value 2

Long. Returns the value of *stringorblob* as a long if it succeeds and *0* if *stringorblob* is not a valid PowerScript number.

Usage

Use Syntax 1 to pass values to external C functions or to specify a value for the LongParm attribute of PowerBuilder's Message object.

To distinguish between a string whose value is the number 0 and a string whose value is not a number when using Syntax 2, use the IsNumber function before calling the Long function.

Examples

These statements convert the UnsignedIntegers nLow and nHigh into a long value:

```
UnsignedInt nLow      //Low integer 16 bits
UnsignedInt nHigh     //High integer 16 bits
long LValue           //Long value 32 bits

nLow = 12345
nHigh = 0
LValue = Long(nLow, nHigh)
MessageBox("Long Value", LValue)
```

This statement returns 2167899876 as a long:

```
Long("2167899876")
```

Syntax 2

After assigning blob data from the database to lb_blob, the following example obtains the long value stored at position 20 in the blob:

```
long lb_num
lb_num = Long(BlobMid(lb_blob, 20, 4))
```

For an example of assigning and extracting values from a blob, see Real.

See also

Dec
Double
Integer
Real
Long in Chapter 2, "DataWindow Painter Functions"

Lower

Description

Converts all the characters in a string to lowercase.

Syntax

Lower (*string*)

Parameter	Description
string	The string you want to convert to lowercase letters

Return value

String. Returns *string* with uppercase letters changed to lowercase if it succeeds and the empty string ("") if an error occurs.

Example

This statement returns babe ruth:

```
Lower("Babe Ruth")
```

See also

Upper
Lower in Chapter 2, "DataWindow Painter Functions"

LowerBound

Description

Obtains the lower bound of a dimension of an array.

Syntax

LowerBound (*array* {, *n*})

Parameter	Description
array	The name of the array for which you want the lower bound of a dimension.
n (optional)	The number of the dimension for which you want the lower bound. The default is 1.

Return value

Integer. Returns the lower bound of dimension *n* of *array* and *-1* if *n* is greater than the number of dimensions of the array.

Usage

For variable-size arrays, memory is allocated for the array when you assign values to it. Before you assign values, the lower bound is 1 and the upper bound is 0.

Examples

The following statements illustrate the values LowerBound reports for fixed-size arrays and for variable-size arrays before and after memory has been allocated:

```
integer a[5], b[2,5]
LowerBound(a)       // Returns 1
LowerBound(a, 1)    // Returns 1
LowerBound(a, 2)    // Returns -1, a has only 1 dim.
LowerBound(b, 2)    // Returns 1
```

```
integer c[ ]
LowerBound(c)        // Returns 1
c[50] = 900
LowerBound(c)        // Returns 1

integer d[-10 to 50]
LowerBound(d)        // Returns -10
```

See also UpperBound

mailAddress

Description Updates the mailRecipient array for a mail message.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to mailSession object

Syntax *mailsession*.**mailAddress** ({ *mailmessage* })

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to address the message.
mailmessage (optional)	A mailMessage structure containing information about the message. If you omit *mailmessage*, mailAddress displays the post office address list.

Return value mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnUserAbort!

Usage

The mailRecipient array contains information about recipients of a mail message or the originator of a message. The originator is not used when you send a message.

If there is an error in the mailRecipient array, mailAddress displays the Address dialog box so the user can fix the address. If you pass a mailMessage structure that is a validly addressed message, for example, a message that the user received, nothing happens because the addresses are correct.

If you don't specify a mailMessage, the mail system displays an Address dialog box that allows users to look for addresses and maintain their personal address list. The user can't select addresses for addressing a message.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Examples

These statements create a mail session, send mail with an attached TXT file and then log off the mail system and destroy the mail session object:

```
mailSession  mSes
mailReturnCode mRet
mailMessage  mMsg
mailFileDescription mAttach

// Create a mail session.
mSes = CREATE mailSession

// Log on to the session
mRet = mSes.mailLogon(mailNewSession!)
IF mRet <> mailReturnSuccess! THEN
   MessageBox ("Mail", 'Logon failed.')
   RETURN
END IF
mMsg.AttachmentFile[1] = mAttach
mRet = mSes.mailAddress(mMsg)
IF mRet <> mailReturnSuccess! THEN
   MessageBox ("Mail", 'Addressing failed.')
   RETURN
END IF

// Send the mail.
mRet = mSes.mailSend(mMsg)
```

```
IF mRet <> mailReturnSuccess! THEN
   MessageBox ("Mail", 'Sending mail failed.')
   RETURN
END IF

mSes.mailLogoff()

DESTROY mSes
```

See also

mailLogoff
mailLogon
mailResolveRecipient
mailSend

mailDeleteMessage

Description

Deletes a mail message from the user's electronic mail inbox.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailDeleteMessage** (*messageid*)

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to delete the message
messageid	A string whose value is the ID of the mail message to be deleted

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnInvalidMessage!
- mailReturnUserAbort!

Usage

To get a list of message IDs in the user's inbox, call the mailGetMessages function.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

Assume the DataWindow dw_inbox contains a list of mail items (sender, subject, postmark, and message ID), and that the mail session mSes has been created and a successful logon has occurred. This script for the clicked event for dw_inbox deletes the selected message from the mail system:

```
string sID
integer  nRow
mailReturnCode mRet

nRow = GetClickedRow()
IF nRow > 0 THEN
   sID = GetItemString(nRow, "messageID")
   mRet = mSes.mailDeleteMessage(sID)
END IF
```

See also

mailGetMessages
mailLogon

mailGetMessages

Description

Populates the messageID array of a mailSession object with the message IDs in the user's inbox.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailGetMessages** ({ *messagetype*, } { *unreadonly* })

Parameter	Description
mailsession	A mail session object identifying the session in which you want to get the messages.
messagetype (optional)	A string whose value is a message type. The default message type is IPM or an empty string (""), which identifies interpersonal messages. The other standard type is IPC, which identifies hidden, interprocess messages. Your mail administrator may have established other user-defined message types.
unreadonly (optional)	A boolean value indicating you want only the IDs of unread messages. Values are: ♦ TRUE — Get IDs for unread messages only ♦ FALSE — Get IDs for all messages

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnNoMessages!
- mailReturnUserAbort!

Usage

MailGetMessages only retrieves message IDs, which it stores in the mailSession object's MessageID array. A message ID serves as an argument for other mail functions. For example, with mailReadMessage, it identifies the message you want to read.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

This example populates a DataWindow with the messages in the user's inbox. The DataWindow is defined with an external data source and has three columns: msgid, msgdate, and msgsubject. MailGetMessages fills the MessageID array in the mailSession object and mailReadMessage gets the information for each ID. The example assumes that the application has already created the mailSession object mSes and logged on:

```
mailMessage msg
integer n
long c_row

mSes.mailGetMessages( )

FOR n = 1 to UpperBound(mSes.MessageID[])
   mSes.mailReadMessage(mSes.MessageID[n],  &
      msg, mailEnvelopeOnly!, FALSE)

   c_row = dw_1.InsertRow( 0 )
   dw_1.SetItem(c_row, "msgid", mSes.MessageID[n])
   dw_1.SetItem(c_row, "msgdate", msg.DateReceived)
   // Truncate subject to fit defined column size
   dw_1.SetItem(c_row, "msgsubject",  &
      Left(msg.Subject, 50))
NEXT
```

See also

mailDeleteMessage
mailReadMessage

mailHandle

Description

Obtains the handle of a mailSession object.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailHandle**( )

Parameter	Description
mailsession	A mailSession object identifying the session for which you want the handle

Return value

UnsignedLong. Returns the internal handle of the mail session object.

Usage

After you have logged on, your mailSession has a valid handle. You can use that handle to call external mail functions. MAPI has additional functions that PowerBuilder doesn't implement directly.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

This statement returns the handle of the current mail session:

```
current_session.mailHandle( )
```

mailLogoff

Description

Ends the mail session, breaking the connection between the PowerBuilder application and mail. If the mail application was already running when PowerBuilder began the mail session, it is left in the same state.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailLogoff** ( )

Parameter	Description
mailsession	A mailSession object identifying the session from which you want to log off

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!

Usage

To release the memory used by the mailSession object, use the DESTROY keyword after ending the mail session.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

This statement terminates the current mail session:

```
current_session.mailLogoff( )
DESTROY current_session
```

See also

mailLogon

mailLogon

Description

Establishes a mail session for the PowerBuilder application. The PowerBuilder application can start a new session or join an existing session.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailLogon**({*userid*, *password*} {, *logonoption*})

Parameter	Description
mailsession	A mailSession object identifying the session you want to logon to.
userid (optional)	A string whose value is the user's mail system user ID.
password (optional)	A string whose value is the user's mail system password.

Parameter	Description
logonoption (optional)	A value of the mailLogonOption enumerated data type specifying the logon options: ♦ mailNewSession! — Starts a new mail session, whether or not the mail application is already running. ♦ mailDownLoad! — Forces the mail application to download any new messages from the server to the user's inbox. Starts a new mail session only if the mail application is not running. ♦ mailNewSessionWithDownLoad! — Starts a new mail session and forces new messages to be downloaded from the server to the user's inbox. The default is to use an existing session if possible and not to force new messages to be downloaded.

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnLoginFailure!
- mailReturnInsufficientMemory!
- mailReturnTooManySessions!
- mailReturnUserAbort!

Usage

If you don't direct mailLogon to start a new session and the mail application is already running on the user's computer, then the PowerBuilder mail session "piggybacks" on the existing session. A user ID and password are not necessary.

When mailLogon establishes a new session, then the mail system's dialog box prompts for the user ID and password if the script doesn't supply them.

The download option forces the mail server to download the latest messages to the user's inbox. This ensures that the inbox is up-to-date; it does not make the messages available to PowerBuilder. To access messages, use mailGetMessages and mailReadMessage.

Before calling mailLogon, you must declare and create a mailSession object.

Examples

In this example, the mailSession object new_session is an instance variable of the window. The window's Open event script allocates memory for the mailSession object and logs on. During the logon process, the mail application displays a dialog box prompting for the user ID and password:

```
new_session = CREATE mailSession
new_session.mailLogon(mailNewSession!)
```

This example establishes a new mail session and makes the user's inbox up-to-date. The user won't be prompted for an ID and password because user information is provided. Here, the mailSession object is a local variable:

```
mailSession new_session
new_session = CREATE mailSession
new_session.mailLogon("jpl", "hotstuff",  &
   mailNewSessionWithDownLoad!)
```

See also

mailLogoff

mailReadMessage

Description

Opens a mail message whose ID is stored in the mail session's message array. You can choose to read the entire message or the envelope (sender, date received, etc.) only. If a message has attachments, they are stored in a temporary file. You can also choose to have the message text written to in a temporary file.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailReadMessage** (*messageid*, &
mailmessage, *readoption*, *mark*)

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to read a message.

Parameter	Description
messageid	A string whose value is the ID of the mail message you want to read.
mailmessage	A mailMessage structure in which mailReadMessage stores the message information.
readoption	A value of the mailReadOption enumerated data type: ♦ mailEntireMessage! — Obtain header, text, and attachments ♦ mailEnvelopeOnly! — Obtain header information only ♦ mailBodyAsFile! — Obtain header, text, and attachments, and treat the message text as the first attachment, storing it in a temporary file ♦ mailSuppressAttach! — Obtain header and text, but no attachments
mark	A boolean indicating whether you want to mark the message as read in the user's inbox. Values are: ♦ TRUE — Mark the message as read ♦ FALSE — Do not mark the message as read

Return value

MailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!

Usage

To obtain the message IDs for the messages in the user's inbox, call mailGetMessages.

> **Reading attachments**
> If a message has an attachment and you don't suppress attachments, information about it is stored in the AttachmentFile attribute of the mailMessage object. The AttachmentFile attribute is a mailFileDescription object. Its PathName attribute has the location of the temporary file that mailReadMessage created for the attachment. By default, the temporary file is in the directory specified by the TEMP environment variable.
>
> Be sure to delete this temporary file when you no longer need it.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

In this example, mail is displayed in a window with a DataWindow dw_inbox that lists mail messages and a MultiLineEdit mle_note that displays the message text. Assume that the application has created the mailSession object mSes and successfully logged on, and that dw_inbox contains a list of mail items (sender, subject, postmark, and message ID). This script for the Clicked event for dw_inbox displays the text of the selected message in the MultiLineEdit mle_note:

```
integer  nRow, nRet
string sMessageID
string sRet, sName


// Find out what Mail Item was selected
nRow = GetClickedRow( )
IF nRow > 0 THEN
   // Get the message ID from the row
   sMessageID = GetItemString(nRow, 'MessageID')

   //Re-read the message to obtain entire  contents
   // because previously we read only the envelope
   mRet = mSes.mailReadMessage(sMessageID, mMsg  &
      mailEntireMessage!, TRUE)

   // Display the text
   mle_note.Text = mMsg.NoteText
END IF
```

See mailGetMessages for an example that creates a list of mail messages in a DataWindow control, the type of setup that this example expects.

See also the mail examples in the samples that are supplied with PowerBuilder.

See also

File functions
mailGetMessages
mailLogon
mailSend

mailRecipientDetails

Description

Displays a dialog box with the specified recipient's address information.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailRecipientDetails** (*mailrecipient* &
{, *allowupdates* })

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to display the detail information for a recipient.
mailrecipient	A mailRecipient structure containing valid address information. *Mailrecipient* must contain a recipient identifier returned by mailAddress, mailResolveRecipient, or mailReadMessage.
allowupdates (optional)	A boolean indicating whether updates to the recipient's name will be allowed. If the user doesn't have update privileges for the mail system, then *allowupdates* is ignored. The default is FALSE.

Return value

MailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailUnknownReturnRecipient!
- mailUnknownReturnUserAbort!

Usage

The effect of setting *allowupdates* to TRUE depends on the mail system and the user's privileges.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

This example gets the message IDs from the user's inbox and reads the first message. It then calls mailRecipientDetails to display address information for the first recipient. . Recipient is an array of structures and an attribute of mailMessage. Each array element is one of the message's recipients. The example does not check how many values there are in the message ID or recipient arrays and it assumes that the application has already created a mailSession object and logged on:

```
mailMessage msg
integer n
long c_row

mSes.mailGetMessages( )
mSes.mailReadMessage(mSes.MessageID[1],  &
   msg, mailEnvelopeOnly!, FALSE )
mSes.mailRecipientDetails(msg.Recipient[1])
```

See also

mailResolveRecipient
mailSend

mailResolveRecipient

Description

Obtains a valid email address based on a partial or full user name and optionally updates information in the system's address list if the user has privileges to do so.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailResolveRecipient** (*recipient* {, *allowupdates* })

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to resolve the recipient.

Parameter	Description
recipient	A mailRecipient structure or a string variable whose value is a recipient's name. The recipient's name is an attribute of the mailRecipient structure. MailResolveRecipient sets the value of the string to the recipient's full name or the structure to the resolved address information.
allowupdates (optional)	A boolean indicating whether updates to the recipient's name will be allowed. If the user doesn't have update privileges for the mail system, then *allowupdates* is ignored. The default is FALSE.

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnUserAbort!

Usage

Use mailResolveRecipient to verify that a name is a valid address in the mail system. The function reports mailReturnFailure! if the name is not found.

If you supply a mailRecipient structure, mailResolveRecipient fills the structure with valid address information when it resolves the address. If you supply a name as a string, mailResolveRecipient replaces the string's value with the full user name as recognized by the mail system. An address specified as a string is adequate for users in the local mail system. If you are sending mail through gateways to other systems, you should obtain full address details in a mailRecipient structure.

If more than one address on the mail system matches the partial address information you supply to mailResolveRecipient, the mail system may display a dialog box allowing the user to choose the desired name.

If you supply a mailRecipient structure that already has address information, mailResolveRecipient will correct the information if it differs from the mail system. If you set *allowupdates* to TRUE and the information differs from the mail system, mailResolveRecipient will correct the *mail system's* information if the user has rights to do so. Be careful that the address information you have is correct when you allow updating.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Examples

This example checks whether there is a user J Smith is on the mail system. If there is a user whose name matches, for example, Jane Smith or Jerry Smith, the variable mname is set to the full name. If both names are on the system, the mail system displays a dialog box from which the user chooses a name. Mname is set to the user's choice. The application has already created the mailSession object mSes and logged on:

```
mailReturnCode mRet
string mname
mname = "Smith, J"
mRet = mSes.mailResolveRecipient(mname)
IF mRet = mailReturnSuccess! THEN
   MessageBox("Address", mname + " found.")
ELSEIF mRet = mailReturnFailure! THEN
   MessageBox("Address", "J Smith not found.")
ELSE
   MessageBox("Address", "Request not evaluated.")
END IF
```

In this example, sle_to contains the full or partial name of a mail recipient. This example assigns the name to a mailRecipient object and calls mailResolveRecipient to find the name and get address details. If the name is found, mailRecipientDetails displays the information and the full name is assigned to sle_to. The application has already created the mailSession object mSes and logged on:

```
mailReturnCode mRet
mailRecipient mRecip

mRecip.Name = sle_to.Text
mRet = mSes.mailResolveRecipient(mRecip)
IF mRet <> mailReturnSuccess! THEN
   MessageBox ("Address", sle_to.Text + "not
found.")
ELSE
   mRet = mSes.mailRecipientDetails(mRecipient)
   sle_to.Text = mRecipient.Name
END IF
```

See also

mailAddress
mailLogoff
mailLogon
mailRecipientDetails
mailSend

mailSaveMessage

Description

Creates a new message in the user's inbox or replaces an existing message.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailSaveMessage** (*messageid*, *mailmessage*)

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to save the mail message.
messageid	A string whose value is the message ID of the message being replaced. If you are saving a new message, specify an empty string ("").
mailmessage	A mailMessage structure containing the message being saved.

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnInvalidMessage!
- mailReturnUserAbort!
- mailReturnDiskFull!

Usage

Before saving a message, you must address the message even if you are replacing an existing message. The message can be addressed to someone else for sending later.

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Examples

This example creates a new message in the inbox of the current user, which will be sent later to Jerry Smith. The application has already created the mailSession object mSes and logged on:

```
mailRecipient recip
mailMessage msg
mailReturnCode mRet

recip.Name = "Smith, Jerry"
mRet = mSes.mailResolveRecipient( recip )
IF mRet <> mailReturnSuccess! THEN  &
   MessageBox("Save New Message",  &
   "Invalid address.")

msg.NoteText = mle_note.Text
msg.Subject = sle_subject.Text
msg.Recipient[1] = recip

mRet = mSes.mailSaveMessage("", msg)
IF mRet <> mailReturnSuccess! THEN  &
   MessageBox("Save New Message",  &
   "Failed somehow.")
```

This example replaces the last message in the user Jane Smith's inbox. It gets the message ID from the MessageID array in the mailSession object mSes. It changes the message subject, re-addresses the message to the user, and saves the message. The application has already created the mailSession object mSes and logged on:

```
mailRecipient recip
mailMessage msg
mailReturnCode mRet
string s_ID

mRet = mSes.mailGetMessages( )
IF mRet <> mailReturnSuccess! THEN
   MessageBox("No Messages", "Inbox empty.")
   RETURN
END IF
s_ID = mSes.MessageID[UpperBound(mSes.MessageID)]
mRet = mSes.mailReadMessage(s, msg,  &
   mailEntireMessage!, FALSE )
IF mRet <> mailReturnSuccess! THEN
   MessageBox("Message", "Can't read message.")
   RETURN
END IF

msg.Subject = msg.Subject + " Test"
recip.Name = "Smith, Jane"
mRet = mSes.mailResolveRecipient( recip )
msg.Recipient[1] = recip

mRet = mSes.mailSaveMessage(s_ID, msg)
IF mRet <> mailReturnSuccess! THEN  &
   MessageBox("Save Old Message", "Failed somehow.")
```

See also the mail examples in the samples that are supplied with PowerBuilder.

See also

mailReadMessage
mailResolveRecipient

mailSend

Description

Sends a mail message. If no message information is supplied, the mail system provides a dialog box for entering it before sending the message.

> **Platform information**
> The mail functions have no effect on the Macintosh.

Applies to

mailSession object

Syntax

mailsession.**mailSend** ({ *mailmessage* })

Parameter	Description
mailsession	A mailSession object identifying the session in which you want to send the mail message
mailmessage (optional)	A mailMessage structure

Return value

mailReturnCode. Returns one of the following values:

- mailReturnSuccess!
- mailReturnFailure!
- mailReturnInsufficientMemory!
- mailReturnLogFailure!
- mailReturnUserAbort!
- mailReturnDiskFull!
- mailReturnTooManySessions!
- mailReturnTooManyFiles!
- mailReturnTooManyRecipients!

- mailReturnUnknownRecipient!
- mailReturnAttachmentNotFound!

Usage

Before calling mail functions, you must declare and create a mailSession object and call mailLogon to establish a mail session.

Example

These statements create a mail session, send a message, and then log off the mail system and destroy the mail session object:

```
mailSession  mSes
mailReturnCode  mRet
mailMessage  mMsg

// Create a mail session
mSes = create mailSession

// Log on to the session
mRet = mSes.mailLogon(mailNewSession!)
IF mRet <> mailReturnSuccess! THEN
   MessageBox ("Mail", 'Logon failed.')
   RETURN
END IF

// Populate the mailFileDescription structure
mMsg.Subject = mle_subject.Text
mMsg.NoteText = 'Luncheon at 12:15'
mMsg.Recipient[1].name = 'Smith, John'
mMsg.Recipient[2].name = 'Shaw, Sue'

// Send the mail
mRet = mSes.mailSend ( mMsg )

IF mRet <> mailReturnSuccess! THEN
   MessageBox("Mail Send", 'Mail not sent')
   RETURN
END IF

mSes.mailLogoff()
DESTROY mSes
```

See also the mail examples in the samples that are supplied with PowerBuilder.

See also

mailReadMessage
mailResolveRecipient

Match

Description

Determines whether a string's value contains a particular pattern of characters.

Syntax

Match (*string, textpattern*)

Parameter	Description
string	The string in which you want to look for a pattern of characters
textpattern	A string whose value is the text pattern

Return value

Boolean. Returns TRUE if *string* matches *textpattern* and FALSE if it does not. Match also returns FALSE if either argument has not been assigned a value or the pattern is invalid.

Usage

Match enables you to evaluate whether a string contains a general pattern of characters. To find out whether a string contains a specific substring, use the Pos function.

Textpattern is similar to a regular expression. It consists of metacharacters, which have special meaning, and ordinary characters, which match themselves. You can specify that the string begin or end with one or more characters from a set, or that it contain any characters except those in a set. See Chapter 3, "Using Text Patterns" for a list of metacharacters and their meaning, as well as sample patterns.

Examples

This statement returns TRUE if the text in sle_ID begins with one or more uppercase or lowercase letters (^ at the beginning of the pattern means that the beginning of the string must match the characters that follow):

```
Match(sle_ID.Text, "^[A-Za-z]")
```

This statement returns FALSE if the text in sle_ID contains any digits (^ inside a bracket is a complement operator):

```
Match(sle_ID.Text, "[^0-9]")
```

This statement returns TRUE if the text in sle_ID contains one uppercase letter:

```
Match(sle_ID.Text, "[A-Z]")
```

This statement returns TRUE if the text in sle_ID contains one or more uppercase letters (+ indicates one or more occurrences of the pattern):

```
Match(sle_ID.Text, "[A-Z]+")
```

This statement returns FALSE if the text in sle_ID contains anything other than two digits followed by a letter (^ and $ indicate the beginning and end of the string):

```
Match(sle_ID.Text, "^[0-9][0-9][A-Za-z]$")
```

See also

Pos
Match in Chapter 2, "DataWindow Painter Functions"
Chapter 3, "Using Text Patterns"

Max

Description

Determines the larger of two numbers.

Syntax

Max (*x*, *y*)

Parameter	Description
x	The number to which you want to compare *y*
y	The number to which you want to compare *x*

Return value

The data type of *x* or *y*, whichever data type is more precise.

Usage

If either of the values being compared is NULL, Max returns NULL.

Examples

This statement returns 7:

```
Max(4,7)
```

This statement returns -4:

```
Max(-4,-7)
```

This statement returns 8.2, a decimal value:

```
Max(8.2, 4)
```

See also

Min
Max in Chapter 2, "DataWindow Painter Functions"

MemberDelete

Description

Deletes a member from an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberDelete** (*membername*)

Parameter	Description
olestorage	The name of an object variable of type OLEStorage containing the member (substorage or stream) you want to delete.
membername	A string specifying the name of the member you want to delete from the storage.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The storage is not open
- *-2* Member not found
- *-3* Insufficient resources or too many files open
- *-4* Access denied
- *-5* Invalid storage state
- *-9* Other error

Examples

This example creates a storage object and opens an OLE object in a file. It checks whether wordobj is a substorage within that object and, if so, deletes it and saves the object back to the file:

```
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
   result = stg_stuff.MemberDelete("wordobj")
   IF result = 0 THEN stg_stuff.Save( )
END IF
```

See also

MemberExists
MemberRename
Open

MemberExists

Description

Determines whether the named member is part of an OLE object in a storage. The member can be another OLE object (a substorage) or a stream.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberExists** (*membername*, *exists*)

Parameter	Description
olestorage	The name of an object variable of type OLEStorage that you want to check
membername	A string whose value is the name of the member that you want to check
exists	A boolean variable that will store whether or not the member exists

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The storage is not open
- *-9* Other error

Examples

This example creates a storage object and opens an OLE object in a file. It checks whether wordobj is a substorage within that object and, if so, deletes it and saves the object back to the file:

```
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
   result = stg_stuff.MemberDelete("wordobj")
   IF result = 0 THEN stg_stuff.Save( )
END IF
```

See also

MemberDelete
MemberRename
Open

MemberRename

Description

Renames a member in an OLE storage. The member can be another OLE object (a substorage) or a stream.

Applies to

OLEStorage objects

Syntax

olestorage.**MemberDelete** (*membername*, *newname*)

Parameter	Description
olestorage	The name of an object variable of type OLEStorage containing the member (substorage or stream) you want to rename
membername	A string whose value is the name of the member you want to rename
newname	A string whose value is the new name to be assigned to the member

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The storage is not open
- *-2* Member not found
- *-3* Insufficient resources or too many files open
- *-4* Access denied
- *-5* Invalid storage state
- *-6* Duplicate name
- *-9* Other error

Examples

This example creates a storage object and opens an OLE object in a file. It checks whether wordobj is a substorage within that object and, if so, renames it to memo and saves the object back to the file:

```
boolean lb_memexists
integer result

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

stg_stuff.MemberExists("wordobj", lb_memexists)
IF lb_memexists THEN
   result =  &
      stg_stuff.MemberRename("wordobj", "memo")
   IF result = 0 THEN stg_stuff.Save( )
END IF
```

See also

MemberDelete
MemberExists
Open

MessageBox

Description

Displays a system MessageBox with the title, text, icon, and buttons you specify.

Syntax

MessageBox (*title, text* {, *icon* {, *button* {, *default*} } })

Parameter	Description
title	A string specifying the title of the message box, which appears in the box's title bar.
text	The text you want to display in the message box. The text can be a numeric data type, a string, or a boolean value.
icon (optional)	A value of the Icon enumerated data type indicating the icon you want to display on the left side of the message box. Values are: ♦ Information! (Default) ♦ StopSign! ♦ Exclamation! ♦ Question! ♦ None!
button (optional)	A value of the Button enumerated data type indicating the set of CommandButtons you want to display at the bottom of the message box. The buttons are numbered in the order listed in the enumerated data type. Values are: ♦ OK! — (Default) OK button ♦ OKCancel! — OK and Cancel buttons ♦ YesNo! — Yes and No buttons ♦ YesNoCancel! — Yes, No, and Cancel buttons ♦ RetryCancel! — Retry and Cancel buttons ♦ AbortRetryIgnore! — Abort, Retry, and Ignore buttons
default (optional)	The number of the button you want to be the default button. The default is 1. If you specify a number larger than the number of buttons displayed, MessageBox uses the default.

Return value

Integer. Returns the number of the selected button (*1*, *2*, or *3*) if it succeeds and *-1* if an error occurs.

Usage

If the value of *title* or *text* is NULL, the MessageBox does not display.

Unless you specify otherwise, PowerBuilder continues executing the script when the user clicks the button or presses ENTER, which is appropriate when the MessageBox has one button. If the box has multiple buttons, you will need to include code in the script that checks the return value and takes an appropriate action.

Before continuing with the current application, the user must respond to the MessageBox. However, the user can switch to another application without responding to the MessageBox.

When MessageBox doesn't work (Windows 3.1 only)
Controls capture the mouse in order to perform certain operations. For instance, CommandButtons capture during mouse clicks, Edit controls capture for text selection, and scrollbars capture during scrolling. If a MessageBox is invoked while the mouse is captured, Windows becomes unstable. Therefore PowerBuilder won't display a MessageBox in the ScrollVertical and ScrollHorizontal events and in similar situations. Instead, you can display text in a text control or in the window's title. (Note: Although the mouse is captured during clicking, the click is complete when the Clicked event script is run, so MessageBox works.)

Because MessageBox grabs focus, you should not use it when focus is changing, such as in a LoseFocus event. Instead, you might display a message in the window's title or a MultiLineEdit.

MessageBox also causes confusing behavior when called after PrintOpen. See PrintOpen for details.

Examples

This statement displays a MessageBox with the title Greeting, the text Hello User, the default icon (Information!), and the default button (the OK button):

```
MessageBox("Greeting", "Hello User")
```

The following statements display a MessageBox titled Result and containing the result of a function, the Exclamation icon, and the OK and Cancel buttons (the Cancel button is the default):

```
integer Net
long Distance = 3.457

Net = MessageBox("Result", Abs(Distance),  &
      Exclamation!, OKCancel!, 2)
IF Net = 1 THEN
   . . .  // Process OK.
ELSE
   . . .  // Process CANCEL.
END IF
```

Mid

Description

Obtains a specified number of characters from a specified position in a string.

Syntax

Mid (*string*, *start* {, *length*})

Parameter	Description
string	The string from which you want characters returned.
start	A long specifying the position of the first character you want returned. (The position of the first character of the string is 1.)
length (optional)	A long whose value is the number of characters you want returned. If you do not enter *length* or if *length* is greater than the number of characters to the right of *start*, Mid returns the remaining characters in the string.

Return value

String. Returns characters specified in *length* of *string* starting at character *start*. If *start* is greater than the number of characters in *string*, the Mid function returns the empty string (""). If l*ength* is greater than the number of characters remaining after the *start* character, Mid returns the remaining characters. The return string is not filled with spaces to make it the specified length.

Usage

To search a string for the position of the substring that you want to extract, use the Pos function. Use the return value for the *start* argument of Mid.

To extract a specified number of characters from the beginning or end of a string, use the Left or the Right function.

Examples

This statement returns RUTH:

```
Mid("BABE RUTH", 5, 5)
```

This statement returns "":

```
Mid("BABE RUTH", 40, 5)
```

This statement returns BE RUTH:

```
Mid("BABE RUTH", 3)
```

These statements store the characters in the SingleLineEdit sle_address from the 40th character to the end in ls_address_extra:

```
string address_extra
ls_address_extra = Mid(sle_address.Text, 40)
```

The following user-defined function, called str_to_int_array, converts a string into an array of integers. Each integer in the array will contain two characters (one characters as the high byte (ASCII value * 256) and the second character as the low byte). The function arguments are str, a string passed by value, and iarr[ ], an integer array passed by reference. The length of the array is initialized before the function is called. If the integer array is longer than the string, the script stores spaces. If the string is longer, the script ignores the extra characters:

To call the function, use code like the following:

```
int rtn
iarr[20]=0     // Initialize the array, if necessary
rtn = str_to_int_array("This is a test.", iarr)
```

The str_to_int_array function is:

```
long stringlen
integer arraylen, i
string char1, char2

// Get the string and array lengths.
arraylen = UpperBound(iarr)
stringlen = Len(str)

// Loop through the array.
FOR i = 1 to arraylen
   IF (i*2 <= stringlen) THEN
      // Get two chars from str
      char1 = Mid(str, i*2, 1)
      char2 = Mid(str, i*2 - 1, 1)

   ELSEIF (i*2 - 1 <= stringlen) THEN
      // Get the last char
      char1 = " "
      char2 = Mid(str, i*2 - 1, 1)

   ELSE
      // Use spaces if beyond the end of str
      char1 = " "
      char2 = " "
   END IF

   iarr[i] = Asc(char1) * 256 + Asc(char2)
NEXT
RETURN 1
```

For sample code that converts the integer array back to a string, see Asc.

See also

Asc
Left
Pos
Right
UpperBound
Mid in Chapter 2, "DataWindow Painter Functions"

Min

Description

Determines the smaller of two numbers.

Syntax

Min (*x*, *y*)

Parameter	Description
x	The number to which you want to compare *y*
y	The number to which you want to compare *x*

Return value

The data type of *x* or *y*, whichever data type is more precise.

Usage

If either of the values being compared is NULL, Min returns NULL.

Examples

This statement returns 4:

```
Min(4,7)
```

This statement returns -7:

```
Min(-4,-7)
```

This statement returns 3.0, a decimal value:

```
Min(9.2,3.0)
```

See also

Max
Min in Chapter 2, "DataWindow Painter Functions"

Minute

Description

Obtains the number of minutes in the minutes portion of a time value.

Syntax

Minute (*time*)

Parameter	Description
time	The time value from which you want the minutes

Return value

Integer. Returns the minutes portion of *time* (00 to 59).

Example

This statement returns 1:

```
Minute(19:01:31)
```

See also

Hour
Second
Minute in Chapter 2, "DataWindow Painter Functions"

Mod

Description

Obtains the remainder (modulus) of a division operation.

Syntax

Mod (*x*, *y*)

Parameter	Description
x	The number you want to divide by *y*
y	The number you want to divide into *x*

Return value

The data type of *x* or *y*, whichever data type is more precise.

Examples

This statement returns 2:

```
Mod(20, 6)
```

This statement returns 1.5:

```
Mod(25.5, 4)
```

This statement returns 2.5:

```
Mod(25, 4.5)
```

See also Mod in Chapter 2, "DataWindow Painter Functions"

ModifiedCount

Description Reports the number rows that have been modified but not updated in a DataWindow.

Applies to DataWindow controls and child DataWindows

Syntax *datawindowname*.**ModifiedCount** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of rows that have been modified but not updated in the associated database table

Return value Long. Returns the number of rows that have been modified in the primary buffer. Returns *0* if no rows have been modified or if all modified rows have been updated in the database table. Returns *-1* if an error occurs.

Usage ModifiedCount reports the number of modified rows in the primary buffer. If a row is modified and then filtered out, PowerBuilder no longer includes it in the count. If the data is filtered again so that the row returns to the primary buffer, PowerBuilder then includes it in the count.

The DeletedCount function counts the number of rows in the deleted buffer. The RowCount function counts the total number of rows in the primary buffer.

Examples

If five rows in dw_Employee have been modified but not updated in the associated database table or filtered out of the primary buffer, the following code sets ll_Rows equal to 5:

```
long ll_Rows
ll_Rows = dw_Employee.ModifiedCount( )
```

If any rows in dw_Employee have been modified but not updated in the associated database table, this statement updates the database table associated with the dw_employee DataWindow control:

```
IF dw_employee.ModifiedCount( ) > 0 THEN  &
      dw_employee.Update()
```

See also

DeleteRow
DeletedCount
FilteredCount
Retrieve
RowCount
Update

Modify

Description

Modifies a DataWindow object by applying specifications, specified as a list of instructions, that change the DataWindow object's definition. You can change appearance, behavior, and database information for the DataWindow object by changing the values of attributes. You can add and remove objects from the DataWindow object by providing specifications for the objects.

For lists and explanations of DataWindow object attributes, see Appendix A. For syntax for creating new objects, see Appendix B.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Modify** (*modstring*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you are modifying.
modstring	A string whose value is the specifications for the modification. See Usage for appropriate formats.

Return value

String. Returns the empty string ("") if it succeeds and an error message if an error occurs. The error message takes the form "Line *n* Column *n* incorrect syntax". The columns are counted from the beginning of the compiled text of *modstring*.

Usage

Modify lets you make many of the same settings in a script that you would make in the DataWindow painter. Typical uses for Modify are:

- Changing colors, text settings, and other appearance settings of objects
- Changing the update status of different tables in the DataWindow so that you can update more than one table
- Modifying the WHERE clause of the DataWindow's SQL SELECT statement
- Turning on Query mode or Prompt For Criteria so users can specify the data they want
- Changing the status of Retrieve Only As Needed
- Changing the data source of the DataWindow object
- Controlling the Print Preview display
- Deleting and adding objects, for example, lines or bitmaps, in the DataWindow object

Each of these uses is illustrated in the examples for this function.

You can use three types of statements in *modstring* to modify a DataWindow object:

Statement type	Use to
CREATE *object(settings)*	Adds *object* to the DataWindow object (such as text, computed fields, and bitmaps). *Settings* is a list of attributes and values using the format in Appendix B, "DataWindow Syntax Listing." To create an object, you must supply enough information to define it.
DESTROY [COLUMN] *object*	Removes *object* from the DataWindow object. When *object* is a column name, specify the keyword COLUMN to remove both the column and the column's data from the buffer.
objectname.attribute=value	Changes the value of *attribute* to *value*. Attributes control the location, color, size, font, and other settings for *objectname*. When *objectname* is DataWindow, you can also set attributes for database access. Appendix A, "DataWindow Object Attributes," lists objects in a DataWindow, their attributes, and values. Depending on the specific attribute, *value* can be: ♦ A constant ♦ A quoted constant ♦ An expression that consists of a default value followed by a valid DataWindow expression that returns the appropriate data type for the attribute. Expressions are described below.

Object names

The DataWindow painter automatically gives names to columns and column labels. Other objects that you add to the DataWindow object are named with a cryptic string of numbers unless you give them names. (Describe will report the cryptic names, but you can't see them in the painter.) To easily describe and modify attributes of an object, give the object a name.

Expressions for Modify

When you specify an expression for a DataWindow attribute, the expression has the format:

defaultvalue~tDataWindowpainterexpression

Defaultvalue is a value that can be converted to the appropriate data type for the attribute. It is followed by a tab (~t). *DataWindowpainterexpression* is an expression that can use any DataWindow painter function. The expression must also evaluate to the appropriate data type for the attribute. When you are setting a column's attribute, the expression is evaluated for each row in the DataWindow, which allows you to vary the display based on the data. A typical expression uses the If function:

```
'16777215 ~t If(emp_status=~~'A~~',255,16777215)'
```

To use that expression in a modstring, specify the following (entered as a single line):

```
modstring = "emp_id.Color='16777215 ~t
If(emp_status=~~'A~~',255,16777215)'"
```

Not all attributes accept expressions. See Appendix A for details on each attribute.

Quotes and tildes

Because Modify's argument is a string, which can include other strings, you need to use special syntax to specify quotation marks. To specify that a quotation mark be used within the string rather than match and end a previously opened quote, you can either specify the other style of quote (single quotes nested with double quotes) or precede the quotation mark with a tilde (~).For another level of nesting, the string itself must specify ~", so you must specify ~~ (which specifies a tilde) followed by ~" (which specifies a quote). For example, another way to type the modstring shown above is (entered as a single line):

```
modstring = "emp_id.Color=~"16777215 ~t
If(emp_status=~~~"A~~~",255,16777215)~""
```

For more information about quotes and tildes, see *PowerScript Language*.

Building a modstring with variables

To use variable data in *modstring*, you can build the string using variables in your program. As you concatenate sections of *modstring*, make sure quotes are included in the string where necessary. For example, the following code builds a modstring similar to the one above, but the default color value and the two color values in the If function are calculated in the script. Notice how the single quotes around the expression are included in the first and last pieces of the string:

```
red_amount = Integer(sle_1.Text)
modstring = "emp_id.Color='" + &
   String(RGB(red_amount, 255, 255)) + &
   "~tIf(emp_status=~~'A~~'," + &
   String(RGB(255, 0, 0)) + &
   "," + &
   String(RGB(red_amount, 255, 255)) + &
   ")'"
```

The following is a simpler example without the If function. You don't need quotes around the value if you are not specifying an expression. Here the String and RGB functions result in a constant value in the resulting modstring:

```
modstring = "emp_id.Color=" +  &
   String(RGB(red_amount, 255, 255))
```

You can set several attributes with a single call to Modify by including each attribute setting in *modstring* separated by spaces. For example, assume the following is entered on a single line in the script editor:

```
rtn = dw_1.Modify("emp_id.Font.Italic=0
oval_1.Background.Mode=0
oval_1.Background.Color=255")
```

However, it is easier to understand and debug a script in which each call to Modify sets one attribute.

Debugging tip

If you build your *modstring* and store it in a variable that is the argument for Modify, you can look at the value of the variable in Debug mode. When Modify's error message reports a column number, you can count the characters as you look at the compiled *modstring*.

Modifying a WHERE clause

Use Modify instead of SetSQLSelect to modify a WHERE clause. Modify verifies the syntax only once and does not change the update status of the DataWindow object. SetSQLSelect modifies the syntax twice (when the syntax is modified and when the retrieve executes) and affects the update status of the DataWindow object.

PowerBuilder already includes many functions for modifying a DataWindow. Before using Modify, check the list of DataWindow functions in *Objects and Controls* to see if a function exists for making the change. Many of these functions are listed in See also below.

Modify is for modifying the attributes of a DataWindow *object*. You can set attributes of the DataWindow *control* that contains the object using standard dot notation. For example, to put a border on the control, specify:

```
dw_1.Border = TRUE
```

Examples

These examples illustrate the typical uses listed in the Usage section.

Changing colors

The effect of setting the Color attribute depends on the object you are modifying. To set the background color of the whole DataWindow object, use the following syntax:

dwcontrolname.**Modify**("DataWindow.Color='*long*'")

To set the text color of a column or a text object, use similar syntax:

dwcontrolname.**Modify**("*objectname*.Color='*long*'")

To set the background color of a column or other object, use the following syntax to set the mode and color. Make sure the mode is opaque:

dwcontrolname.**Modify**(&
"*objectname*.Background.Mode='<0 - Opaque, 1 - Transparent>'")

dwcontrolname.**Modify**("*objectname*.Background.Color='*long*'")

The following examples use the syntaxes shown above to set the colors of various parts of the DataWindow object.

This statement changes the background color of the DataWindow dw_cust to red:

```
dw_cust.Modify("DataWindow.Color = 255")
```

This statement causes the DataWindow dw_cust to display the text of values in the salary column in red if they exceed 90000 and in green if they do not:

```
dw_cust.Modify(  &
    "salary.Color='0~tIf(salary>90000,255,65280)'")
```

This statement nests one If function within another to provide three possible colors. The setting causes the DataWindow dw_cust to display the department ID in green if the ID is 200, in red if it is 100, and in black if it is neither:

```
dw_cust.Modify("dept_id.Color='0~t "&
   + "If(dept_id=200,65380,If(dept_id=100,0,255))'")
```

The following example uses a complex expression with nested If functions to set the background color of the salary column according to the salary values. Each portion of the concatenated string is shown on a separate line. See the pseudocode in the comments for an explanation of what the nested If functions do. The example also sets the background mode to opaque so that the color settings are visible. The example includes error checking, which displays Modify's error message, if any:

```
string mod_string, err
long color1, color2, color3, default_color

err = dw_emp.Modify("salary.Background.Mode=0")
IF err <> "" THEN
   MessageBox("Status",  &
      "Change to Background Mode Failed " + err)
   RETURN
END IF

/* Pseudocode for mod_string:
If salary less than 10000, set the background to
red.
If salary greater than or equal to 10000 but less
than 20000, set the background to blue.
If salary greater than or equal to 20000 but less
than 30000, set the background color to green.
Otherwise, set the background color to white, which
is also the default.
*/
color1 = 255                //red
color2 = 16711680           //blue
color3 = 65280              //green
default_color = 16777215    //white

mod_string = &
"salary.Background.Color = '"  &
   + String(default_color)  &
   + "~tIf(salary < 10000,"  &
   + String(color1)  &
   + ",If(salary < 20000,"  &
   + String(color2)  &
   + ",If(salary < 30000,"  &
   + String(color3)  &
   + ","  &
   + String(default_color)  &
   + ")))'"

err = dw_emp.Modify(mod_string)
IF err <> "" THEN
   MessageBox("Status", &
      "Change to Background Color Failed " + err)
   RETURN
END IF
```

The following example sets the text color of a RadioButton column to the value of color1, which is red, if the column's value is "Y". Otherwise the text is set to black. As above, each portion of the concatenated string is shown on a separate line:

```
integer color1, default_color
string mod_string, err

color1 = 255            //red
default_color = 0       //black
```

```
mod_string = "yes_or_no.Color ='"  &
   + String(default_color)  &
   + "~tif(yes_or_no=~~'Y~~',"  &
   + String(color1)  &
   + ","  &
   + String(default_color)  &
   + ")'"
err = dw_emp.Modify(mod_string)
IF err <> "" THEN
   MessageBox("Status",  &
   "Modify to Text Color "  &
   + "of yes_or_no Failed " + err)
   RETURN
END IF
```

Changing displayed text

To set the text of a text object, the next two examples use this syntax:

dwcontrolname.**Modify**("*textobjectname*.Text='string'")

This statement changes the text in the text object Dept_t in the DataWindow dw_cust to Dept:

```
dw_cust.Modify("Dept_t.Text='Dept'")
```

This statement sets the displayed text of dept_t in the DataWindow dw_cust to Marketing if the department ID is greater than 201; otherwise it sets the text to Finance:

```
dw_cust.Modify("dept_t.Text='none~t "  &
   + "If(dept_id > 201,~'Marketing~',~'Finance~')'")
```

Updating more than one table

An important use of Modify is to make it possible to update more than one table from one DataWindow object. The following script updates the table that was specified as updatable in the DataWindow painter, then it uses Modify to make the other joined table updatable and to specify the key column and which columns to update. This technique eliminates the need to create multiple DataWindow objects or to use embedded SQL statements to update more than one table.

In the example, the DataWindow object joins two tables: department and employee. First department is updated, with status flags not reset. Then employee is made updatable and is updated. If all succeeds, the Update commands resets the flags and COMMIT commits the changes. Note that to make the script repeatable in the user's session, you must add code to make department the updatable table again:

```
integer rc
string err

/* The SELECT statement for the DataWindow is:
SELECT department.dept_id, department.dept_name,
employee.emp_id, employee.emp_fname,
employee.emp_lname FROM department, employee ;
*/

// Update department, as set up in the DW painter
rc = dw_1.Update(TRUE, FALSE)

IF rc = 1 THEN
   //Turn off update for department columns.
   dw_1.Modify("department_dept_name.Update = No")
   dw_1.Modify("department_dept_id.Update = No")
   dw_1.Modify("department_dept_id.Key = No")

   // Make employee table updatable.
   dw_1.Modify(  &
      "DataWindow.Table.UpdateTable = ~"employee~"")

   //Turn on update for desired employee columns.
   dw_1.Modify("employee_emp_id.Update = Yes")
   dw_1.Modify("employee_emp_fname.Update = Yes")
   dw_1.Modify("employee_emp_lname.Update = Yes")
   dw_1.Modify("employee_emp_id.Key = Yes")

   //Then update the employee table.
   rc = dw_1.Update()
   IF rc = 1 THEN
      COMMIT USING SQLCA;
   ELSE
      ROLLBACK USING SQLCA;
      MessageBox("Status",  &
      + "Update of employee table failed. "  &
      + "Rolling back all changes.")
   END IF
ELSE
   ROLLBACK USING SQLCA;
   MessageBox("Status",  &
      + "Update of department table failed. "  &
      + "Rolling back changes to department.")
END IF
```

Adding a WHERE clause

The following scripts dynamically add a WHERE clause to a DataWindow object that was created with a SELECT statement that did not include a WHERE clause. (Since this example appends a WHERE clause to the original SELECT statement, additional code would be needed to remove a where clause from the original SELECT statement if it had one.) This technique is useful when the arguments in the WHERE clause may change at execution time.

The original SELECT statement might be:

```
SELECT employee.emp_id, employee.l_name
   FROM employee
```

Presumably, the application builds a WHERE clause based on the user's choices. The WHERE clause might be:

```
WHERE emp_id > 40000
```

The script for the window's Open event stores the original SELECT statement in original_select, an instance variable:

```
dw_emp.SetTransObject(SQLCA)
original_select =  &
   dw_emp.Describe("DataWindow.Table.Select")
```

The script for a CommandButton's Clicked event attaches a WHERE clause stored in the instance variable where_clause to original_select and assigns it to the DataWindow's Table.Select attribute:

```
string rc, mod_string

mod_string = "DataWindow.Table.Select='"  &
   + original_select + where_clause + "'"

rc = dw_emp.Modify(mod_string)
IF rc = "" THEN
   dw_emp.Retrieve( )
ELSE
   MessageBox("Status", "Modify Failed" + rc)
END IF
```

Quotes inserted in the DataWindow painter

For Watcom, ORACLE, and ALLBASE, the DataWindow painter puts double quotes around the table and column name (for example, SELECT "EMPLOYEE"."EMP_LNAME"). Unless you have removed the quotes, the sample WHERE clause must also use these quotes. For example:

```
where_clause =  &
" where ~~~"EMPLOYEE~~~".~~~"SALARY~~~" > 40000"
```

Note that owner names are not enclosed in double quotes.

Query mode and prompt for criteria

Query mode provides an alternate view of a DataWindow in which the user specifies conditions for selecting data. PowerBuilder builds the WHERE clause based on the specifications. When the user exits query mode, you can retrieve data based on the modified SELECT statement.

In this example, a window that displays a DataWindow control has a menu that includes a MenuItem called Select Data. When the user chooses it, its script displays the DataWindow control in query mode and checks the MenuItem. When the user chooses it again, the script turns query mode off and retrieves data based on the new WHERE clause specified by the user via query mode. The script also makes a CheckBox labeled Sort data visible, which turns query sort mode on and off.

The script for the Select Data MenuItem is:

```
string rtn

IF m_selectdata.Checked = FALSE THEN
   // Turn on query mode so user can specify data
   rtn = dw_1.Modify("DataWindow.QueryMode=YES")

   IF rtn = "" THEN
      // If Modify succeeds, check MenuItem showing
      // Query mode is on and display sort CheckBox
      This.Check()
      ParentWindow.cbx_sort.Show()
   ELSE
      MessageBox("Error",  &
      "Can't access query mode to select data.")
   END IF
ELSE
   // Turn off Query mode and retrieve data
   // based on user's choices
   rtn = dw_1.Modify("DataWindow.QueryMode=NO")

   IF rtn = "" THEN
      // If Modify succeeds, uncheck MenuItem
      // showing Query mode is off, hide the sort
      // CheckBox, and retrieve data
      This.UnCheck()
      ParentWindow.cbx_sort.Hide()
      dw_1.Retrieve()
   ELSE
      MessageBox("Error",  &
      "Failure exiting query mode.")
   END IF
END IF
```

A simple version of the script for Clicked event of the Sort data CheckBox follows. You could add code as shown in the MenuItem script above to check whether Modify succeeded:

```
IF This.Checked = TRUE THEN
   dw_1.Modify("DataWindow.QuerySort=YES")
ELSE
   dw_1.Modify("DataWindow.QuerySort=NO")
END IF
```

For details on how you or the user specifies information in query mode, see the *User's Guide*.

"Prompt for criteria" is another way of letting the user specify retrieval criteria. You set it on a column-by-column basis. When a script retrieves data, PowerBuilder displays the Specify Retrieval Criteria window, which gives the user a chance to specify criteria for all columns that have been set.

In a script that is run before you retrieve data, for example, in the Open event of the window that displays the DataWindow control, the following settings would make the columns emp_name, emp_salary, and dept_id available in the Specify Retrieval Criteria dialog when the Retrieve function is called:

```
dw_1.Modify("emp_name.Criteria.Dialog=YES")
dw_1.Modify("emp_salary.Criteria.Dialog=YES")
dw_1.Modify("dept_id.Criteria.Dialog=YES")
```

There are other Criteria attributes that affect both query mode and prompt for criteria. See Criteria in Appendix A for details.

Retrieve as needed

In this example, the DataWindow object has been set up with Retrieve Only As Needed selected. When this is on, PowerBuilder retrieves enough rows to fill the DataWindow, displays them quickly, then waits for the user to try to display additional rows before retrieving more rows. If you want the fast initial display but do not want to leave the cursor open on the server, you can turn off Retrieve Only As Needed with Modify.

After you have determined that enough rows have been retrieved, the following code in the RetrieveRow event script changes the Retrieve.AsNeeded attribute, which forces the rest of the rows to be retrieved:

```
dw_1.Modify("DataWindow.Retrieve.AsNeeded=NO")
```

Changing the data source

This example changes the data source of a DataWindow object from a SQL SELECT statement to a stored procedure. This technique works *only* if the result set does not change (that is, the number, type, and order of columns is the same for both sources).

When you define the DataWindow object, you must predefine all possible DataWindow retrieval arguments. In this example, the SELECT statement defined in the painter has three arguments, one of type string, one of type number, and one of type date. The stored procedure has two arguments, both of type string. So, in the painter, you need to define four DataWindow arguments, two of type string, one of type number, and one of type date. (Note that you do not have to use all the arguments you define):

```
string rc, mod_string, name_str = "Watson"
integer dept_num = 100

// Remove the DataWindow's SELECT statement
Dw_1.Modify("DataWindow.Table.Select = ''")

// Set the Procedure attribute to your procedure
mod_string = "DataWindow.Table.Procedure = &
   '1 execute dbo.emp_arg2;1 @dept_id_arg &
   = :num_arg1, @lname_arg = :str_arg1'"
rc = dw_1.Modify(mod_string)

// If change is accepted, retrieve data
IF rc = "" THEN
   dw_1.Retrieve(dept_num, name_str)
ELSE
   MessageBox("Status",  &
   "Change to DW Source Failed " + rc)
END IF
```

Deleting and adding objects in the DataWindow object

This statement deletes a bitmap object called logo from the DataWindow dw_cust:

```
dw_cust.Modify("destroy logo")
```

This statement deletes the column named salary from the DataWindow dw_cust. Note that this example includes the keyword column so the column in the DataWindow and the data are both deleted:

```
dw_cust.Modify("destroy column salary")
```

This example adds a rectangle named rect1 to the header area of the DataWindow dw_cust. (Enter the value of modstring as a single line):

```
string modstring

modstring = 'create rectangle(Band=background
X="206" Y="6" height="69" width="1363"
brush.hatch="6" brush.color="12632256" pen.style="0"
pen.width="14" pen.color="268435584"
background.mode="2" background.color="-1879048064"
name=rect1 )'

dw_cust.Modify(modstring)
```

These statements add a bitmap named logo to the header area for grouping level 1 in the DataWindow dw_cust. (Enter the value of modstring as a single line):

```
string modstring

modstring = 'create bitmap(band=footer x="37" y="12"
height="101" width="1509"
filename="C:\PB3\BEACH.BMP" border="0" name=bmp1 )'

dw_cust.Modify(modstring)
```

Syntax for creating objects
To create an object you must provide DataWindow syntax, as shown in Appendix B. The easiest way to get correct syntax for all the necessary attributes is to paint the object in the DataWindow painter and export the syntax to a file. Then you make any desired changes and put the syntax in your script, as shown above. This is really the only way to get accurate syntax for complex objects like graphs.

See also

Describe
Reset
SetBorderStyle
SetDataStyle
SetFilter
SetFormat
SetPosition
SetRowFocusIndicator
SetSeriesStyle
SetSQLPreview
SetSQLSelect
SetTabOrder
SetValidate
SyntaxFromSQL

ModifyData

Description

Changes the value of a data point in a series on a graph. There are two syntaxes depending on the type of graph:

- For all graph types except scatter, use Syntax 1, in which you can specify the data point to be modified by position or by category.
- For scatter graphs, use Syntax 2, in which you specify the data point by position and provide an x and y value.

Applies to

Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

controlname.**ModifyData** (*seriesnumber*, *datapoint*, & *datavalue* {, *categoryvalue* })

Parameter	Description
controlname	The name of the graph in which you want to modify data.
seriesnumber	The number of the series in which you want to modify data.
datapoint	The number of the data point for which you want to modify the data.
datavalue	The new value of the data point. The data type of *datavalue* is the same as the data type of the values axis of the graph.
categoryvalue (optional)	The category for *datavalue*. The data type of *categoryvalue* is the same as the data type of the category axis of the graph.

Syntax 2

controlname.**ModifyData** (*seriesnumber*, *datapoint*, *xvalue*, *yvalue*)

Parameter	Description
controlname	The name of the scatter graph in which you want to modify data in a series
seriesnumber	The number that identifies the series in which you want to modify data
datapoint	The number of the data point for which you want to modify data
xvalue	The new x value of the data you want to modify
yvalue	The new y value of the data you want to modify

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

For Syntax 1, when you specify *categoryvalue*, ModifyData changes the category value at the specified position, as well as the data value. If the name you specify already exists at another position, the data at that position is modified instead and the position in *datapoint* is ignored (the same behavior as InsertData).

When you specify a position of 0, ModifyData always behaves the same as InsertData.

For scatter graphs (Syntax 2), there are no categories. You specify the position in the series whose data you want to modify and provide the x and y values for the data.

For a comparison of AddData, InsertData, and ModifyData, see Equivalent syntax in InsertData.

Examples

These statements change the data for Apr in the series named Costs in the graph gr_product_data:

```
integer SeriesNbr, CategoryNbr

// Get the number of the series.
SeriesNbr = gr_product_data.FindSeries("Costs")
CategoryNbr = gr_product_data.FindCategory("Apr")
gr_product_data.ModifyData(SeriesNbr, &
   CategoryNbr, 1250)
```

Syntax 2

These statements modify the data point 9 in the series named Test One in the scatter graph gr_product_data:

```
integer SeriesNbr
SeriesNbr = gr_product.FindSeries("Test One")
gr_product_data.ModifyData(SeriesNbr,  &
   9, 4.55, 86.38)
```

See also

AddData
FindCategory
FindSeries
InsertCategory
InsertData

Month

Description

Determines the month of a date value.

Syntax

Month (*date*)

Parameter	Description
date	The date from which you want the month

Return value

Integer. Returns an integer (1 to 12) whose value is the month portion of *date*.

Examples

This statement returns 1:

```
Month(1994-01-31)
```

These statements store in start_month the month entered in the SingleLineEdit sle_start_date:

```
integer start_month
start_month = Month(date(sle_start_date.Text))
```

See also

Day
Date
Year
Month in Chapter 2, "DataWindow Painter Functions"

Move

Description

Moves a control or object to another position relative to its parent window, or for some window objects, relative to the screen.

Applies to

Any object or control, except a line control

Syntax

objectname.**Move** (*x*, *y*)

Parameter	Description
objectname	The name of the object or control you want to move to a new location
x	The x coordinate of the new location in PowerBuilder units
y	The y coordinate of the new location in PowerBuilder units

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

The x and y coordinates you specify are the new coordinates of the upper-left corner of the object or control. If the shape of the object or control is not rectangular (for example, a RadioButton or Oval), x and y are the coordinates of the upper-left corner of the box enclosing it.

When you move controls, drawing objects, and child windows, the coordinates you specify are relative to the upper-left corner of the parent window. When you use Move to position main, popup, and response windows, the coordinates you specify are relative to the upper-left corner of the display screen.

You cannot use Move to move a line control because it has multiple x and y coordinates.

You can specify coordinates outside the frame of the parent window or screen, which effectively makes the object or control invisible.

To draw the image of a Picture control at a particular position, without actually moving the control, use the Draw function.

The Move function changes the X and Y attributes of the moved object.

Equivalent syntax

The syntax below directly sets the X and Y attributes of an object or control. Although the result is equivalent to using the Move function, it causes PowerBuilder to redraw *objectname* twice; first at the new location of X and then at the new X and Y location:

objectname.**X** = *x*
objectname.**Y** = *y*

These statements cause PowerBuilder to redraw gb_box1 twice:

```
gb_box1.X = 150
gb_box1.Y = 200
```

This statement has the same result but redraws gb_box1 once:

```
gb_box1.Move(150,200)
```

Examples

This statement changes the X and Y attributes of gb_box1 to 150 and 200, respectively, and moves gb_box1 to the new location:

```
gb_box1.Move(150, 200)
```

This statement moves the picture p_Train2 next to the picture p_Train1:

```
P_Train2.Move(P_Train1.X + P_Train1.Width, &
      P_Train1.Y)
```

Now

Description

Obtains the current time based on the system time of the client machine.

Syntax

Now ( )

Return value

Time. Returns the current time based on the system time of the client machine.

Usage

Use Now to compare a time to the system time or to display the system time on the screen. You can use the Timer function to trigger a Timer event which causes Now to refresh the display.

Examples

This statement returns the current system time:

```
Now( )
```

This example displays the current time in the StaticText st_time. It keeps the time up-to-date by setting a timer that triggers a Timer event every 60 seconds. Code in the window's Open event displays the initial time and starts the timer. Code in the Timer event displays the time again.

The following code appears in the window's Open event script:

```
st_time.Text = String(Now( ), "hh:mm")
Timer(60)
```

A single line in the Timer event script refreshes the time display:

```
st_time.Text = String(Now( ), "hh:mm")
```

See also

Today
Now in Chapter 2, "DataWindow Painter Functions"

ObjectAtPointer

Description

Finds out where the user clicked in a graph. ObjectAtPointer reports the region of the graph under the pointer and stores the associated series and data point numbers in the designated variables.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**ObjectAtPointer**({*graphcontrol*,} *seriesnumber*, & *datapoint*)

Parameter	Description
controlname	The name of the graph object for which you want the object under the pointer, or the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the object under the pointer
seriesnumber	An integer variable in which you want to store the number of the series under the pointer
datapoint	An integer variable in which you want to store the number of the data point under the pointer

Return value

grObjectType. Returns a value of the grObjectType enumerated data type if the user clicks anywhere in the graph (including an empty area) and a NULL value if the user clicks outside the graph.

Values of grObjectType and the parts of the graph associated with them are:

- TypeCategory! — A label for a category
- TypeCategoryAxis! — The category axis or between the category labels
- TypeCategoryLabel! — The label of the category axis
- TypeData! — A data point or other data marker
- TypeGraph! — Any place within the graph control that isn't another grObjectType
- TypeLegend! — Within the legend box, but not on a series label
- TypeSeries! — The line that connects the data points of a series when the graph's type is line or on the series label in the legend box
- TypeSeriesAxis! — The series axis of a 3D graph
- TypeSeriesLabel! — The label of the series axis of a 3D graph

- TypeTitle! — The title of the graph
- TypeValueAxis! — The value axis, including on the value labels
- TypeValueLabel! — The user clicked the label of the value axis

Usage

The ObjectAtPointer function allows you to find out how the user is interacting with the graph. The function returns a value of the grObjectType enumerated data type identifying the part of the graph. When the user clicks in a series, data point, or category, ObjectAtPointer stores the series and/or data point numbers in designated variables.

When the user clicks a data point (or other data mark, such as line or bar), or on the series labels in the legend, ObjectAtPointer stores the series number in the designated variable. When the user clicks on a data point or category tickmark label, ObjectAtPointer stores the data point number in the designated variable.

When the user clicks in a series, but not on the actual data point, ObjectAtPointer stores 0 in *datapoint* and when the user clicks in a category, ObjectAtPointer stores 0 in *seriesnumber*. When the user clicks other parts of the graph, ObjectAtPointer stores 0 in both variables.

Tip
ObjectAtPointer is most effective as the first function call in the script for the Clicked event for the graph control. Make sure you enable the graph control (the default is disabled). Otherwise, the Clicked event script is never run.

Examples

These statements store the series number and data point number at the pointer location in the graph named gr_product in SeriesNbr and ItemNbr. If the object type is TypeSeries! they obtain the series name, and if it is TypeData! they get the data value:

```
integer SeriesNbr, ItemNbr
double data_value
grObjectType object_type
string SeriesName

object_type = &
   gr_product.ObjectAtPointer(SeriesNbr, ItemNbr)

IF object_type = TypeSeries! THEN
   SeriesName = &
      gr_product.SeriesName(SeriesNbr)
```

```
ELSEIF object_type = TypeData! THEN
   data_value = &
      gr_product.GetData(SeriesNbr, ItemNbr)
END IF
```

These statements store the series number and data point number at the pointer location in the graph named gr_computers in the DataWindow control dw_equipment in SeriesNbr and ItemNbr:

```
integer SeriesNbr, ItemNbr
dw_equipment.ObjectAtPointer("gr_computers", &
   SeriesNbr, ItemNbr)
```

See also

AddData
AddSeries

OLEActivate

Description

Activates Object Linking and Embedding (OLE) for the specified object and sends the specified command verb to the OLE server application.

> **Platform information**
> This and other OLE functions have no effect on the Macintosh.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**OLEActivate** (*row, column, verb*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow from which you want to activate OLE.
row	A long identifying the row location of the OLE object.
column	The column location of the OLE object. *Column* can be a column number (integer) or a column name (string).
verb	Usually 0, but the verb is dependent on the OLE server.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The user can activate OLE by double-clicking on an OLE blob column in a DataWindow. Use OLEActivate when you want to activate OLE in response to some other event or action, for example, when the user clicks on a button.

The verb you specify determines what action occurs when the OLE server application is invoked. The default verb, 0, generally means you want to edit the document. Each OLE application has its own particular set of supported verbs. You can find out what verbs the application supports by using the advanced interface of the Windows RegEdit utility (run REGEDIT /V).

Data for an OLE application is stored in the database as a Binary/Text Large Object (blob). In Watcom, data type of the database column is long binary. To make the blob accessible to users, use the DataWindow painter to set up the blob column. In the painter, you add an OLE Database Blob object to the DataWindow object and specify the OLE server application in the Database Binary/Text Large Object window. See the *PowerBuilder User's Guide* for setup details.

Example

This statement activates OLE for the OLE object in row 5 of the salary column in DataWindow dw_emp_data. The verb is 0:

```
dw_emp_data.OLEActivate(5, "salary", 0)
```

See also

Activate

Open

Description

Opens a window or an OLE object.

For windows, Open displays a window and makes all its attributes and controls available to scripts. There are two syntaxes:

- To open an instance of a particular window data type, use Syntax 1.
- To allow the application to select the window's data type when the script is executed, use Syntax 2.

For OLE objects, Open loads an OLE object contained in a file or storage into an OLE 2.0 control or storage object variable. The source and the target are now connected for the purposes of saving work. There are several syntaxes:

- To open an OLE object in an file and load it into an OLE 2.0 control, use Syntax 3.
- To open an OLE object in a storage object in memory and load it into an OLE 2.0 control, use Syntax 4.
- To open an OLE object in an OLE storage file and load it into a storage object in memory, use Syntax 5.
- To open an OLE object that is a member of an open OLE storage and load it into a storage object in memory, use Syntax 6.
- To open a stream in an OLE storage object in memory and load it into a stream object, use Syntax 7.

Applies to

(Syntax 1 and 2) Window objects

(Syntax 3 and 4) OLE 2.0 controls

(Syntax 5 and 6) OLE storage objects

(Syntax 7) OLE stream objects

Syntax 1

Open (*windowvar* {, *parent* })

Parameter	Description
windowvar	The name of the window you want to display. You can specify a window object defined in the Window painter (which is a window data type) or a variable of the desired window data type. Open places a reference to the opened window in *windowvar*.
parent (child and popup windows only)	The window you want make the parent of the child or popup window you are opening. If you open a child or popup window and omit *parent*, PowerBuilder associates the window being opened with the currently active window.

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

Open (*windowvar*, *windowtype* {, *parent* })

Parameter	Description
windowvar	A window variable, usually of data type window. Open places a reference to the opened window in *windowvar*.
windowtype	A string whose value is the data type of the window you want to open. The data type of *windowtype* must be the same or a descendant of *windowvar*.
parent (child and popup windows only)	The window you want to make the parent of the child or popup window you are opening. If you open a child or popup window and omit *parent*, PowerBuilder associates the window being opened with the currently active window.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 3

ole2control.**Open** (*OLEsourcefile*)

Parameter	Description
ole2control	The name of the OLE 2.0 control into which you want to load an OLE object.
OLEsourcefile	A string specifying the name of an OLE storage file containing the object. The file must already exist and contain an OLE 2.0 object. *OLEsourcefile* can include a path for the file, as well as path information inside the OLE storage.

Return value 3

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The file is not found or its data has an invalid format
- *-9* Other error

Syntax 4

ole2control.**Open** (*sourcestorage* , *substoragename*)

Parameter	Description
ole2control	The name of the OLE 2.0 control into which you want to load an OLE object.
sourcestorage	The name of an object variable of OLEStorage containing the object you want to load into *ole2control*.

Parameter	Description
substoragename	A string specifying the name of a substorage that contains the desired object within *storagename*.

Return value 4

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-2* The parent storage is not open
- *-9* Other error

Syntax 5

olestorage.**Open** (*OLEsourcefile* {, *readmode* {, *sharemode* }})

Parameter	Description
olestorage	The name of a object variable of type OLEStorage into which you want to load the OLE object.
OLEsourcefile	A string specifying the name of an OLE storage file containing the object. The file must already exist and contain OLE 2.0 objects. *OLEsourcefile* can include the file's path, as well as path information within the storage.
readmode (optional)	A value of the enumerated data type stgReadMode that specifies the type of access you want for *OLEsourcefile*. Values are: ♦ stgReadWrite! — (Default) Read/write access. If the file does not exist, Open creates it. ♦ stgRead! — Read-only access. You can't change *OLEsourcefile*. ♦ stgWrite! — Write access. You can rewrite *OLEsourcefile* but not read its current contents. If the file does not exist, Open creates it.
sharemode (optional)	A value of the enumerated data type stgShareMode that specifies how other attempts, by your own or other applications, to open *OLEsourcefile* will fare. Values are: ♦ stgExclusive! — (Default) No other attempt to open *OLEsourcefile* will succeed. ♦ stgDenyNone! — Any other attempt to open *OLEsourcefile* will succeed.

Parameter	Description
	♦ stgDenyRead! — Other attempts to open *OLEsourcefile* for reading will fail. ♦ stgDenyWrite — Other attempts to open *OLEsourcefile* for writing will fail.

Return value 5

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The file is not an OLE 2.0 storage file
- *-3* The file is not found
- *-9* Other error

Syntax 6

olestorage.**Open** (*substoragename*, *readmode*, *sharemode*, & *sourcestorage*)

Parameter	Description
olestorage	The name of a object variable of type OLEStorage into which you want to load the OLE object.
substoragename	A string specifying the name of the storage member within *sourcestorage* that you want to open. Note the reversed order of the *sourcestorage* and *substoragename* arguments from Syntax 4.
readmode	A value of the enumerated data type stgReadMode that specifies the type of access you want for *substoragename*. Values are: ♦ stgReadWrite! — Read/write access. If the member does not exist, Open creates it. ♦ stgRead! — Read-only access. You can't change *substoragename*. ♦ stgWrite! — Write access. You can rewrite *substoragename* but not read its current contents. If the member does not exist, Open creates it.
sharemode	A value of the enumerated data type stgShareMode that specifies how other attempts, by your own or other applications, to open *substoragename* will fare. Values are: ♦ stgExclusive! — (Default) No other attempt to open *substoragename* will succeed.

Parameter	Description
	♦ stgDenyNone! — Any other attempt to open *substoragename* will succeed. ♦ stgDenyRead! — Other attempts to open *substoragename* for reading will fail. ♦ stgDenyWrite — Other attempts to open *substoragename* for writing will fail.
sourcestorage	An open OLEStorage object containing *substoragename*.

Return value 6

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-2* The parent storage is not open
- *-3* The member is not found (when opened for reading)
- *-9* Other error

Syntax 7

olestream.**Open** (*sourcestorage*, *streamname* &
{, *readmode* {, *sharemode* } })

Parameter	Description
olestream	The name of a object variable of type OLEStream into which you want to load the OLE object.
sourcestorage	An OLE storage that contains the stream to be opened.
streamname	A string specifying the name of the stream within *sourcestorage* that you want to open.
readmode (optional)	A value of the enumerated data type stgReadMode that specifies the type of access you want for *streamname*. Values are: ♦ stgReadWrite! — Read/write access. If *streamname* does not exist, Open creates it. ♦ stgRead! — Read-only access. You can't change *streamname*. ♦ stgWrite! — Write access. You can rewrite *streamname* but not read its current contents. If *streamname* does not exist, Open creates it.

Parameter	Description
sharemode (optional)	A value of the enumerated data type stgShareMode that specifies how other attempts, by your own or other applications, to open *streamname* will fare. Values are: ♦ stgExclusive! — No other attempt to open *streamname* will succeed. ♦ stgDenyNone! — Any other attempt to open *streamname* will succeed. ♦ stgDenyRead! — Other attempts to open *streamname* for reading will fail. ♦ stgDenyWrite — Other attempts to open *streamname* for writing will fail.

Return value 7

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Stream not found
- *-2* Stream already exists
- *-3* Stream is already open
- *-4* Storage not open
- *-5* Access denied
- *-6* Invalid name
- *-9* Other error

Usage

You must open a window before you can access the attributes of the window. If you access the window's attributes before you open it, an execution error will occur.

To reference an open window in scripts, use *windowvar*.

Tip for Syntax 1
If you call Syntax 1 of the Open function twice for the same window, PowerBuilder activates the window twice; it does not open two instances of the window.

When you use Syntax 2, the window object specified in *windowtype* must be the same data type as *windowvar* (the data type includes data types inherited from it). The data type of *windowvar* is usually window, from which all windows are inherited, but it can be any ancestor of *windowtype*. If it is not the same type, an execution error will occur.

Use Syntax 2 to open an array of windows when each window in the array will have a different data type. See the last example, in which the window data types are stored in one array and are used for the *windowtype* argument when each window in another array is opened.

Tips for Syntax 2

When you use Syntax 2, PowerBuilder opens an instance of a window of the data type specified in *windowtype* and places a reference to this instance in the variable *windowvar*.

If *windowtype* is a descendant window, you can only reference attributes, events, functions, or structures that are part of the definition of *windowvar*. For example, if a user event is declared for *windowtype*, you cannot reference it.

The object specified in *windowtype* is not automatically included in your executable application. To include it, you must save it in a PBD file (PowerBuilder dynamic library) that you deliver with you application.

Parent windows for the opened window

Generally, if you are opening a child or a popup window and specify *parent*, the window identified by *parent* is the parent of the opened window (*windowname* or *windowvar*). When a parent window is closed, all its child and popup windows are closed too.

Not all types of windows can be parent windows. Only a window whose borders are not confined within another window can be a parent. A child window or a window opened as a sheet cannot be a parent. If you specify a "confined" window as a parent, PowerBuilder will check its parent, and that window's parent, until it finds a window that it can use as a parent. Therefore if you open a popup window and specify a sheet as its parent, PowerBuilder will make the MDI frame that contains the sheet its parent.

If you don't specify a parent for a child or popup window, the active window becomes the parent. Therefore, if one popup is active and you open another popup, the first popup is the parent, not the main window. When the first popup is closed, PowerBuilder closes the second popup too.

Remember though, that, in an MDI application, the active sheet is not the active window and cannot be the parent. In Windows, it is clear that the MDI frame, not the active sheet, is the active window—its title bar is the active color and it displays the menu. On the Macintosh, it is not obvious because the frame is not visible, but the frame is still the active window.

On the Windows platform, Windows enforces this hierarchy of parent and child windows. On the Macintosh, PowerBuilder implements the hierarchy so that cross-platform applications have the same behavior.

OLE storage files

An OLE storage file is structured like a directory. Each OLE object can contain other OLE objects (substorages) and other data (streams). You can open the members of an OLE storage belonging to a server application if you know the structure of the storage. However, PowerBuilder's functions for manipulating storages are provided so that you can build your own storage files for organizing the OLE objects used in your applications.

The whole file can be an OLE object and substorages within the file can also be OLE objects. More frequently, the structure for a storage file you create is a root level that is not an OLE object but contains independent OLE objects as substorages. Any level in the storage hierarchy can contain OLE objects or be simply a repository for another level of substorages.

Opening nested objects
Because you can specify path information within an OLE storage with a backslash as the separator, you can open a deeply nested object with a single call to Open. However, there is no error checking for the path you specify and if the Open fails, you won't know why. It is strongly recommended that you open each object in the path until you get to the one you want.

Examples

This statement opens an instance of a window named w_employee:

```
Open(w_employee)
```

The following statements open an instance of a window of the type w_employee:

```
w_employee w_to_open
Open(w_to_open)
```

The following code opens an instance of a window of the type child named cw_data and makes w_employee the parent:

```
child cw_data
Open(cw_data, w_employee)
```

The following code opens two windows of type w_emp:

```
w_emp w_e1, w_e2
Open(w_e1)
Open(w_e2)
```

Syntax 2

This example opens a window of the type specified in the string s_w_name and stores the reference to the window in the variable w_to_open. The SELECT statement retrieves data specifying the window type from the database and stores it in s_w_name:

```
window w_to_open
string s_w_name

SELECT next_window INTO :s_w_name FROM routing_table
   WHERE ... ;

Open(w_to_open, s_w_name)
```

This example opens an array of ten windows of the type specified in the string is_w_emp1 and assigns a title to each window in the array. The string is_w_emp1 is an instance variable whose value is a window type:

```
integer n
window win_array[10]

FOR n = 1 to 10
   Open(win_array[n], is_w_emp1)
   win_array[n].title = "Window " + string(n)
NEXT
```

The following statements open four windows. The type of each window is stored in the array w_stock_type. The window reference from the Open function is assigned to elements in the arrow w_stock_win:

```
window w_stock_win[ ]
string w_stock_type[4]

w_stock_type[1] = "w_stock_wine"
w_stock_type[2] = "w_stock_scotch"
w_stock_type[3] = "w_stock_beer"
w_stock_type[4] = "w_stock_soda"

FOR n = 1 to 4
   Open(w_stock_win[n], w_stock_type[n])
NEXT
```

Syntax 3

This example opens the object in the file MYSTUFF.OLE and loads it into in the control ole_1.

```
integer result
result = ole_1.Open("c:\ole2\mystuff.ole")
```

Syntax 4

This example opens the object in the substorage excel_obj within the storage variable stg_stuff and loads it into the control ole_1. Olest_stuff is already open:

```
integer result
result = ole_1.Open(stg_stuff, "excel_obj")
```

This example opens a substorage in the storage variable stg_stuff and loads it into the control ole_1. The substorage name is specified in the variable stuff_1. Olest_stuff is already open:

```
integer result
string stuff_1 = "excel_obj"
result = ole_1.Open(stg_stuff, stuff_1)
```

Syntax 5

This example opens the object in the file MYSTUFF.OLE and loads it into the OLEStorage variable stg_stuff:

```
integer result
OLEStorage stg_stuff

stg_stuff = CREATE OLEStorage
result = stg_stuff.Open("c:\ole2\mystuff.ole")
```

This example opens the same object for reading:

```
integer result
OLEStorage stg_stuff

stg_stuff = CREATE OLEStorage
result = stg_stuff.Open("c:\ole2\mystuff.ole",  &
   stgRead!)
```

Syntax 5 and 6

This example opens the object in the file MYSTUFF.OLE and loads it into the OLEStorage variable stg_stuff, as in the previous example. Then it opens the substorage drawing_1 into a second storage variable. (This example does not include code to close and destroy any of the objects that were opened):

```
integer result
OLEStorage stg_stuff, stg_drawing

stg_stuff = CREATE OLEStorage
result = stg_stuff.Open("c:\ole2\mystuff.ole")
IF result >= 0 THEN
   stg_drawing = CREATE OLEStorage
   result = opest_drawing.Open("drawing_1",  &
      stgRead!, stgDenyNone!, stg_stuff)
END IF
```

Syntax 5 and 7

This example opens the object in the file MYSTUFF.OLE and loads it into the OLEStorage variable stg_stuff. Then it checks whether a stream called info exists in the OLE object, and if so, opens it with read access. (This example does not include code to close and destroy any of the objects that were opened):

```
integer result
boolean str_found
OLEStorage stg_stuff
OLEStream mystream

stg_stuff = CREATE OLEStorage
result = stg_stuff.Open("c:\ole2\mystuff.ole")
IF result < 0 THEN RETURN

result = stg_stuff.MemberExists("info", str_found)
IF result < 0 THEN RETURN

IF str_found THEN
   mystream = CREATE OLEStream
   result = mystream.Open(stg_stuff, "info",  &
      stgRead!, stgDenyNone!)
   IF result < 0 THEN RETURN
END IF
```

See also

Close
InsertFile
OpenWithParm
Save
SaveAs
Show

OpenChannel

Description

Opens a channel to a DDE server application.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

OpenChannel (*applname, topicname* {,*windowhandle* })

Parameter	Description
applname	A string specifying the DDE name of the DDE server application
topicname	A string identifying the data or the instance of the application you want to use (for example, in Microsoft Excel, the topic name could be System or the name of an open spreadsheet)
windowhandle (optional)	The handle of the window that you want to act as the DDE client. Specify this parameter to control which window is acting as the DDE client when you have more than one open window.

Return value

Long. Returns the handle to the channel (a positive integer) if it succeeds. If an error occurs, OpenChannel returns a negative integer. Values are:

- *-1* Open failed
- *-9* handle is NULL

Usage

Use OpenChannel to open a channel to a DDE server application and leave it open so you can efficiently execute more than one DDE request. This type of DDE conversation is called a warm link. Because you open a channel, the operating system does not have to poll all open applications every time you send or ask for data.

The following is an outline of a warm-link conversation:

- Open a DDE channel with OpenChannel and check that it returns a valid channel handle (a positive value).
- Execute several DDE functions. You can use the following functions:

  ExecRemote(*command, handle, <windowhandle>*)

  GetRemote(*location, target, handle, <windowhandle>*)

  SetRemote(*location, value, handle, <windowhandle>*)
- Close the DDE channel with CloseChannel.

If you only need to use a remote DDE function once, you can call ExecRemote, GetRemote, or SetRemote without opening a channel. This is called a cold link. Without an open channel, the operating system polls all running applications to find the specified server application each time you call a DDE function.

Your PowerBuilder application can also be a DDE server. See the StartServerDDE function for more information.

About server applications
Each application decides how it supports DDE. You must check each potential server application's documentation to find out its DDE name, what its valid topics are, and how it expects locations to be specified.

Examples

These statements open a channel to the active spreadsheet REGION.XLS in Microsoft Excel and set handle to the handle to the channel:

```
long handle
handle = OpenChannel("Excel", "REGION.XLS")
```

The following example opens a DDE channel to Excel and requests data from three spreadsheet cells. In the PowerBuilder application, the data is stored in the string array s_regiondata. The client window for the DDE conversation is w_ddewin:

```
long handle
string s_regiondata[3]

handle = OpenChannel("Excel", "REGION.XLS",  &
      Handle(w_ddewin))
GetRemote("R1C2", s_regiondata[1], handle,  &
      Handle(w_ddewin))
GetRemote("R1C3", s_regiondata[2], handle,  &
      Handle(w_ddewin))
GetRemote("R1C4", s_regiondata[3], handle,  &
      Handle(w_ddewin))
CloseChannel(handle, Handle(w_ddewin))
```

See also

CloseChannel
ExecRemote
GetRemote
SetRemote

OpenSheet

Description

Opens a sheet within an MDI (multiple document interface) frame window and creates a menu item for selecting the sheet on the specified menu.

Applies to

Window objects

Syntax

OpenSheet (*sheetrefvar* {, *windowtype* }, *mdiframe* {, *position* & {, *arrangeopen* } })

Parameter	Description
sheetrefvar	The name of any window variable that is not an MDI frame window. OpenSheet places a reference to the open sheet in *sheetrefvar*.
windowtype (optional)	A string whose value is the data type of the window you want to open. The data type of *windowtype* must be the same or a descendant of *sheetrefvar*.
mdiframe	The name of an MDI frame window.
position (optional)	The number of the menu item (in the menu associated with the sheet) to which you want to append the names of the open sheets. Menu bar menu items are numbered from the left, beginning with 1. The default is to list the open sheets under the next-to-last menu item.
arrangeopen (optional)	A value of the ArrangeOrder enumerated data type specifying how you want the sheet arranged in the MDI frame in relation to other sheets when it is opened: ♦ Cascaded! — (Default) Cascade the sheet relative to other open sheets, so that its title bar is below the previously opened sheet. ♦ Layered! — Layer the sheet so that it fills the frame and covers previously opened sheets. ♦ Original! — Open the sheet in its original size and cascade it.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A sheet is a document window that is contained within an MDI frame window. MDI applications allow several sheets to be open at the same time. The newly opened sheet becomes the active sheet. If the opened sheet has an associated menu, that menu becomes the menu at the top of the frame.

When you specify *windowtype*, the window object specified in *windowtype* must be the same data type as *sheetrefvar* (a data type includes data types inherited from it). The data type of *sheetrefvar* is usually window, from which all windows are inherited, but it can be any ancestor of *windowtype*. If it is not the same type, an execution error will occur.

OpenSheet opens a sheet and appends its name to the item on the menu bar specified in *position*. If *position* is 0 or greater than the number of items on the menu bar, PowerBuilder appends the name of the sheet to the next-to-last menu item in the menu bar. In most MDI applications, the next-to-last menu item on the menu bar is the Window menu, which contains options for arranging sheets, as well as the list of open sheets.

> **If the sheets don't appear on the menu**
> PowerBuilder can't append the sheets to a menu that doesn't have any other MenuItems. Make sure that the menu you specify or, if you leave out *position*, the next-to-last menu has at least one other MenuItem.

If more than nine sheets are open in the frame, the first nine are listed on the menu specified by *position* and a final item More Windows is added.

Example

This statement opens the sheet child_1 in the MDI frame MDI_User in its original size. It appends the name of the opened sheet to the second menu item in the menu bar, which is now the menu associated with child_1, not the menu associated with the frame:

```
OpenSheet(child_1, MDI_User, 2, Original!)
```

This example opens an instance of the window object child_1 as an MDI sheet and stores a reference to the opened window in child. The name of the sheet is appended to the fourth menu associated with child_1 and is layered:

```
window child
OpenSheet(child, "child_1", MDI_User, 4, Layered!)
```

See also

ArrangeSheets
GetActiveSheet
OpenSheetWithParm

OpenSheetWithParm

Description

Opens a sheet within an MDI (multiple document interface) frame window and creates a menu item for selecting the sheet on the specified menu, as OpenSheet does. OpenSheetWithParm also stores a parameter in the system's Message object so that it is accessible to the opened sheet.

Applies to

Window objects

Syntax

OpenSheetWithParm (*sheetrefvar, parameter* {, *windowtype*}, & *mdiframe* {, *position* {, *arrangeopen* } })

Parameter	Description
sheetrefvar	The name of any window variable that is not an MDI frame window. OpenSheet places a reference to the open sheet in *sheetrefvar*.
parameter	The parameter you want to store in the Message object when the sheet is opened. *Parameter* must have one of these data types: ♦ String ♦ Numeric ♦ PowerObject
windowtype (optional)	A string whose value is the data type of the window you want to open. The data type of *windowtype* must be the same or a descendant of *sheetrefvar*.
mdiframe	The name of the MDI frame window in which you want to open this sheet.
position (optional)	The number of the menu item (in the menu associated with the sheet) to which you want to append the names of the open sheets. Menu bar menu items are numbered from the left, beginning with 1. The default is to list the open sheets under the next-to-last menu item.

Parameter	Description
arrangeopen (optional)	A value of the ArrangeOrder enumerated data type specifying how you want the sheets arranged in the MDI frame when they are opened: ♦ Cascaded! — (Default) Cascade the sheet relative to other open sheets, so that its title bar is below the previously opened sheet. ♦ Layered! — Layer the sheet so that it fills the frame and covers previously opened sheets. ♦ Original! — Open the sheet in its original size and cascade it.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The system Message object has three attributes for storing data. Depending on the data type of the parameter specified for OpenSheetWithParm, scripts for the opened sheet would check one of the following attributes.

Message object attribute	Parameter data type
Message.DoubleParm	Numeric
Message.PowerObjectParm	PowerObject (PowerBuilder objects, including user-defined structures)
Message.StringParm	String

In the opened window, it is a good idea to access the value passed in the Message object right away because some other script may use the Message object for another purpose.

Avoiding null object references
When you pass a PowerObject as a parameter, you are passing a reference to the object. The object must exist when you refer to it later or you will get a null object reference. For example, if you pass the name of a control on a window that is being closed, that control will not exist when a script accesses the parameter.

See also the usage notes for OpenSheet, which also apply to OpenSheetWithParm.

Example

This statement opens the sheet w_child_1 in the MDI frame MDI_User in its original size and stores MA in message.StringParm. It appends the names of the open sheet to the second menu item in the menu bar of MDI_User (the menu associated with w_child_1):

```
OpenSheetWithParm(w_child_1, "MA",  &
    MDI_User, 2, Original!)
```

The next example illustrates how to access parameters passed in the Message object. These statements are in the scripts for two different windows. The script for the first window declares child as a window and opens an instance of w_child_1 as an MDI sheet. The name of the sheet is appended to the fourth menu item associated with w_child_1 and is layered.

The script also passes a reference to the SingleLineEdit control sle_state as a PowerObject parameter of the Message object. The script for the Open event of w_child_1 uses the text in the edit control to determine what type of calculations to perform. Note that this would fail if sle_state no longer existed when the second script refers to it. As an alternative, you could pass the text itself, which would be stored in the String parameter of Message.

The second script determines the text in the SingleLineEdit and performs processing based on that text.

The script for the first window is:

```
window child
OpenSheetWithParm(child, sle_state,  &
   "w_child_1", MDI_User, 4, Layered!)
```

The second script, for the Open event in w_child_1, is:

```
SingleLineEdit sle_state
sle_state = Message.PowerObjectParm
IF sle_state.Text = "overtime" THEN
   . . .  // overtime hours calculations
ELSEIF sle_state.Text = "vacation" THEN
   . . .  // vacation processing
ELSEIF sle_state.Text = "standard" THEN
   . . .  // standard hours calculations
END IF
```

See also

ArrangeSheets
OpenSheet
OpenSheetWithParm

OpenUserObject

Description

Adds a user object to the specified window and makes all its attributes and controls available to scripts. There are two syntaxes. Syntax 1 opens an instance of a particular user object. Syntax 2 allows the application to select the user object's type when the script is executed.

Applies to

Window objects

Syntax 1

windowname.**OpenUserObject** (*userobjectvar* {, *x*, *y* })

Parameter	Description
windowname	The name of the window in which you want to open the user object.
userobjectvar	The name of the user object you want to display. You can specify a user object defined in the User Object painter (which is a user object data type) or a variable of the desired user object data type. Open places a reference to the opened user object in *userobjectvar*.
x (optional)	The x coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.
y (optional)	The y coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

windowname.**OpenUserObject** (*userobjectvar*, *userobjecttype* & {, *x*, *y* })

Parameter	Description
windowname	The name of the window in which you want to open the user object.
userobjectvar	A variable of data type DragObject. OpenUserObject places a reference to the opened user object in *userobjectvar*.
userobjecttype	A string whose value is the name of the user object you want to display. The data type of *userobjecttype* must be a descendant of *userobjectvar*.
x (optional)	The x coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.

Parameter	Description
y (optional)	The y coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use Syntax 1 when you know what user object you want to open. Use Syntax 2 when the application will determine what type of user object to open when the script runs.

You must open a user object before you can access the attributes of the user object. If you access the user object's attributes before you open it, an execution error will occur.

OpenUserObject *does not* add the user object to the window's Control array, which is an attribute that lists the window's controls.

A user object that is part of a window's definition (that is, it was added to the window in the Window painter) does not have to opened in a script. PowerBuilder opens it when it opens the window.

PowerBuilder displays the user object when it next updates the display or at the end of the script, whichever comes first. For example, if you open several user objects in a script, they will all display at once when the script is complete, unless some other statements cause a change it the screen's appearance (for example, the MessageBox function displays a message or the script changes a visual attribute of a control).

Tip for Syntax 1
If you call Syntax 1 twice to open the same user object, PowerBuilder activates the user object twice; it does not open two instances of the user object.

Tips for Syntax 2

When you use Syntax 2, PowerBuilder opens an instance of a user object of the data type specified in *userobjecttype* and places a reference to this instance in the variable *userobjectname*. To refer to the instance in scripts, use *userobjectname*.

If *userobjecttype* is a descendant user object, you can only refer to attributes, events, functions, or structures that are part of the definition of *userobjectname*. For example, if a user event is declared for *userobjecttype*, you cannot reference it.

The object specified in *userobjecttype* is not automatically included in your executable application. To include it, you must save it in a PBD file (PowerBuilder dynamic library) that you deliver with you application.

Examples

This statement displays an instance of a user object named u_Employee in the upper left corner of the window w_emp (coordinates 0,0):

```
w_emp.OpenUserObject(u_Employee)
```

The following statements display an instance of a user object u_to_open at 200,100 in the window w_empstatus:

```
u_employee u_to_open
w_empstatus.OpenUserObject(u_to_open, 200, 100)
```

The following statement displays an instance of a user object u_data at location 20,100 in w_info:

```
w_info.OpenUserObject(u_data, 20, 100)
```

Syntax 2

The following example displays a user object of the type specified in the string u_name and stores the reference to the user object in the variable u_to_open. The user object is located at 100,200 in the window w_info:

```
DragObject u_to_open
string s_u_name

u_name = sle_user.Text
w_info.OpenUserObject(u_to_open, s_u_name, 100, 200)
```

See also

OpenUserObjectWithParm

OpenUserObjectWithParm

Description

Adds a user object to the specified window and makes all its attributes and controls available to scripts, as OpenUserObject does. OpenUserObjectWithParm also stores a parameter in the system's Message object so that it is accessible to the opened object.

There are two syntaxes. Syntax 1 opens an instance of a particular user object. Syntax 2 allows the application to select the user object's type when the script is executed.

Applies to

Window objects

Syntax 1

windowname.**OpenUserObjectWithParm** (*userobjectvar*, & *parameter* {, *x*, *y* })

Parameter	Description
windowname	The name of the window in which you want to open the user object.
userobjectvar	The name of the user object you want to display. You can specify a user object defined in the User Object painter (which is a user object data type) or a variable of the desired user object data type. OpenUserObject places a reference to the opened user object in *userobjectvar*.
parameter	The parameter you want to store in the Message object when the user object is opened. *Parameter* must have one of these data types: ♦ String ♦ Numeric ♦ PowerObject
x (optional)	The x coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.
y (optional)	The y coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

windowname.**OpenUserObjectWithParm** (*userobjectvar* , & *parameter*, *userobjecttype* {, *x* ,*y* })

Parameter	Description
windowname	The name of the window in which you want to open the user object.
userobjectvar	A variable of data type DragObject. OpenUserObject places a reference to the opened user object in *userobjectvar*.
parameter	The parameter you want to store in the Message object when the user object is opened. *Parameter* must have one of these data types: ♦ String ♦ Numeric ♦ PowerObject
userobjecttype	A string whose value is the data type of the user object you want to open. The data type of *userobjecttype* must be a descendant of *userobjectvar*.
x (optional)	The x coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.
y (optional)	The y coordinate in PowerBuilder units of the user object within the window's frame. The default is 0.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The system Message object has three attributes for storing data. Depending on the data type of the parameter specified for OpenUserObjectWithParm, scripts for the opened user object would check one of the following attributes.

Message object attribute	Parameter data type
message.DoubleParm	Numeric
message.PowerObjectParm	PowerObject (PowerBuilder objects, including user-defined structures)
message.StringParm	String

In the opened user object, it is a good idea to access the value passed in the Message object right away because some other script may use the Message object for another purpose.

Avoiding null object references
When you pass a PowerObject as a parameter, you are passing a reference to the object. The object must exist when you refer to it later or you will get a null object reference. For example, if you pass the name of a control on a window that is being closed, that control will not exist when a script accesses the parameter.

See also the usage notes for OpenUserObject, all of which apply to OpenUserObjectWithParm.

Examples

This statement displays an instance of a user object named u_Employee in the window w_emp and stores the string James Newton in Message.StringParm. The Constructor event script for the user object uses the string parameter as the text of a StaticText control st_empname in the object. The script that opens the user object has the following statement:

```
w_emp.OpenUserObjectWithParm(u_Employee,  &
   "James Newton")
```

The user object's Constructor event script has the following statement:

```
st_empname.Text = Message.StringParm
```

The following statements display an instance of a user object u_to_open in the window w_emp and store a number in message.DoubleParm:

```
u_employee u_to_open
integer age = 50
w_emp.OpenUserObjectWithParm(u_to_open, age)
```

Syntax 2

The following statement displays an instance of a user object u_data of type u_benefit_plan at location 20,100 in the window w_hresource. The parameter "Benefits" is stored in message.StringParm:

```
DragObject u_data
w_hresource.OpenUserObjectWithParm(u_data,  &
   "Benefits", "u_benefit_plan", 20, 100)
```

These statements open a user object of the type specified in the string s_u_name and store the reference to the user object in the variable u_to_open. The script gets the value of s_u_name, the type of user object to open, from the database. The parameter is the text of the SingleLineEdit sle_loc, so it is stored in Message.StringParm. The user object is at the default coordinates 0,0 in the window w_info:

```
DragObject u_to_open
string s_u_name, e_location

e_location = sle_location.Text

SELECT next_userobj INTO :s_u_name
   FROM routing_table
   WHERE ... ;

w_info.OpenUserObjectWithParm(u_to_open,  &
   e_location, s_u_name)
```

The following statements display a user object of the type specified in the string u_name and store the reference to the user object in the variable u_to_open. The parameter is numeric so it is stored in message.DoubleParm. The user object is at the coordinates 100,200 in the window w_emp:

```
userobject u_to_open
integer age = 60
string s_u_name

u_name = sle_user.Text
w_emp.OpenUserObjectWithParm(u_to_open, age,  &
   s_u_name, 100, 200)
```

See also

CloseWithReturn
OpenUserObject
OpenWithParm

OpenWithParm

Description

Displays a window and makes all its attributes and controls available to scripts, as Open does. OpenWithParm also stores a parameter in the system's Message object so that it is accessible to the opened window.

There are two syntaxes. Syntax 1 opens an instance of a particular window data type. Syntax 2 allows the application to select the window's data type when the script is executed.

Applies to

Window objects

Syntax 1

OpenWithParm (*windowvar, parameter* {, *parent* })

Parameter	Description
windowvar	The name of the window you want to display. You can specify a window object defined in the Window painter (which is a window data type) or a variable of the desired window data type. Open places a reference to the open window in *windowvar*.
parameter	The parameter you want to store in the Message object when the window is opened. *Parameter* must have one of these data types: ♦ String ♦ Numeric ♦ PowerObject
parent (child and popup windows only)	The window you want make the parent of the child or popup window you are opening. If you open a child or popup window and omit *parent*, PowerBuilder associates the window being opened with the currently active window.

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

OpenWithParm (*windowvar, parameter , windowtype* {, *parent* })

Parameter	Description
windowvar	A window variable, usually of data type window. Open places a reference to the open window in *windowvar*.
parameter	The parameter you want to store in the Message object when the window is opened. *Parameter* must have one of these data types: ♦ String ♦ Numeric ♦ PowerObject
windowtype	A string whose value is the data type of the window you want to open. The data type of *windowtype* must be the same or a descendant of *windowvar*.
parent (child and popup windows only)	The window you want to make the parent of the child or popup window you are opening. If you open a child or popup window and omit *parent*, PowerBuilder associates the window being opened with the currently active window.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The system Message object has three attributes for storing data. Depending on the data type of the parameter specified for OpenWithParm, your scripts for the opened window would check one of the following attributes.

Message object attribute	Parameter data type
Message.DoubleParm	Numeric
Message.PowerObjectParm	PowerObject (PowerBuilder objects, including user-defined structures)
Message.StringParm	String

In the opened window, it is a good idea to access the value passed in the Message object right away because some other script may use the Message object for another purpose.

Avoiding null object references

When you pass a PowerObject as a parameter, you are passing a reference to the object. The object must exist when you refer to it later or you will get a null object reference. For example, if you pass the name of a control on a window that is being closed, that control will not exist when a script accesses the parameter.

Passing several values as a structure

To pass several values, create a user-defined structure to hold the values and access the PowerObjectParm attribute of the Message object in the opened window. The structure is passed by value, not by reference, so you can access the information even if the original structure has been destroyed.

See also the usage notes for Open, all of which apply to OpenWithParm.

Examples

This statement opens an instance of a window named w_employee and stores the string parameter in Message.StringParm. The script for the window's Open event uses the string parameter as the text of a StaticText control st_empname. The script that opens the window has the following statement:

```
OpenWithParm(w_employee, "James Newton")
```

The window's Open event script has the following statement:

```
st_empname.Text = Message.StringParm
```

The following statements open an instance of a window of the type w_to_open. Since the parameter is a number it is stored in Message.DoubleParm:

```
w_employee w_to_open
integer age = 50
OpenWithParm(w_to_open, age)
```

The following statement opens an instance of a child window named cw_data and makes w_employee the parent. The window w_employee must already be open. The parameter benefit_plan is a string and is stored in Message.StringParm

```
OpenWithParm(cw_data, "benefit_plan", w_employee)
```

Syntax 2

These statements open a window of the type specified in the string s_w_name and store the reference to the window in the variable w_to_open. The script gets the value of s_w_name, the type of window to open, from the database. The parameter in e_location is text, so it is stored in Message.StringParm.

```
window w_to_open
string s_w_name, e_location

e_location = sle_location.Text

SELECT next_window INTO :s_w_name
   FROM routing_table
   WHERE ... ;

OpenWithParm(w_to_open, e_location, s_w_name)
```

The following statements open a window of the type specified in the string c_w_name, store the reference to the window in the variable wc_to_open, and make w_emp the parent window of wc_to_open. The parameter is numeric, so it is stored in Message.DoubleParm:

```
window wc_to_open
string c_w_name
integer age = 60

c_w_name = "w_c_empl"

OpenWithParm(wc_to_open, age, c_w_name, w_emp)
```

See also

CloseWithReturn
Open

ParentWindow

Description

Obtains the parent window of a window.

Applies to

Window objects

Syntax

windowname.**ParentWindow ( )**

Parameter	Description
windowname	The name of a window for which you want to obtain the parent object

Return value

Window. Returns the parent of *windowname*. Returns a null object reference if an error occurs.

Usage

The ParentWindow function, along with the pronoun Parent, allows you to write more general scripts by avoiding the coding of actual window names. Parent refers to the window that contains the current object or control, in other words, the local environment. ParentWindow returns the parent window of a specified window. It gives you information about the "environment" of a window.

Whether a window has a parent depends on its type and how it was opened. You can specify the parent when you open the window. For windows that always have parents, PowerBuilder chooses the parent if you don't specify it. Response windows and child windows always have a parent window. The parent of a sheet in an MDI application is the MDI frame window. Popup windows have a parent window when they are opened from another window. A popup window opened from the application's Open event does not have a parent.

The pronoun ParentWindow is only used in scripts for menu items and refers to the window with which the menu item is associated.

Examples

These statements return the parent of child_1. The parent is a window of the data type Win1:

```
Win1 w_parent
w_parent = child_1.ParentWindow( )
```

The following script for a Cancel button in a popup window triggers an event for the parent window of the button's parent window (the window that contains the button). Then it closes the button's window. The parent window of that window will have a script for the cancelrequested event:

```
Parent.ParentWindow().TriggerEvent  &
   ("cancelrequested")
Close(Parent)
```

Paste

Description

Inserts (pastes) the contents of the clipboard into the specified control. For editable controls, text on the clipboard is pasted at the insertion point. For OLE 2.0 controls, the OLE object on the clipboard replaces any object already in the control.

Applies to

EditMask, MultiLineEdit, SingleLineEdit, DropDownListBox, DataWindow, OLE 2.0 controls

Syntax

controlname.**Paste** ( )

Parameter	Description
controlname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, DropDownListBox or OLE 2.0 control into which you want to insert the contents of the clipboard. If *controlname* is a DataWindow, text is pasted into the edit box over the current row and column. If *controlname* is a DropDownListBox, the AllowEdit attribute must be TRUE.

Return value

Integer.

For edit controls, returns the number of characters that were pasted into *controlname*. If nothing has been cut or copied (the clipboard is empty) or the clipboard contains nontext data (for example, a bitmap or OLE object), the Paste function does not change the contents of the edit control and returns *0*.

For OLE 2.0 controls, returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* No data or clipboard contents is not embeddable
- *-9* Other error

Usage

For editable controls, if text is selected in *controlname*, Paste replaces the text with the contents of the clipboard. If the clipboard contains more lines than fit in the edit control, only the number of lines that fit are pasted.

In a DataWindow control, the text is pasted into the edit control over the current row and column. If the clipboard contains more text that is allowed for that column, the text is truncated. If the clipboard text doesn't match the column's data type, all the text is truncated, so that any selected text is replaced with an empty string.

To insert a specific string in *controlname* or to replace selected text with a specific string, use the ReplaceText function.

When you use Paste to put an OLE object in an OLE 2.0 control, the data is embedded in the PowerBuilder application, not linked.

Examples

If the clipboard contains Proposal good for 90 days and no text is selected, this statement pastes Proposal good for 90 days in mle_Comment1 at the insertion point and returns 25:

```
mle_Comment1.Paste( )
```

If the clipboard contains the string Final Edition, mle_Comment2 contains This is a Preliminary Draft, and the text in mle_Comment2 is selected, this statement deletes This is a Preliminary Draft, replaces it with Final Edition, and returns 13:

```
mle_Comment2.Paste( )
```

If the clipboard contains an OLE object, this statement makes it the contents of the control ole_1 and returns 0:

```
ole_1.Paste( )
```

See also

Copy
Cut
PasteLink
PasteSpecial
ReplaceText

PasteLink

Description

Pastes a link to the contents of the clipboard into the control. The server application for the object on the clipboard must be running.

Applies to

OLE 2.0 controls

Syntax

ole2control.**PasteLink** ( )

Parameter	Description
ole2control	The name of the OLE 2.0 control into which you want to paste the object on the clipboard

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* No data or the contents of the clipboard is not linkable
- *-9* Other error

Usage

When you copy data to the clipboard from an application that supports OLE 2.0 (the server application), you can paste the object into PowerBuilder's OLE 2.0 control with a link to the original data. Object information about the source of the data is only available if the server application is running. You don't need to worry about running the server application if you are working with an OLE object that PowerBuilder knows about, such as an object in a PowerBuilder library or an object that is part of a control's definition in a window. For these objects, PowerBuilder will run the server application in the background to enable the link.

PasteLink will fail, though, if the user switches to a server application, copies the data, quits the application, and then tries to paste and link the object in their PowerBuilder application.

Examples

If the clipboard contains an OLE object and the object's server application is running, then the following example pastes the object in the control ole_1 and sets result to 0:

```
integer result
result = ole_1.PasteLink( )
```

See also

LinkTo
Paste
PasteSpecial

PasteSpecial

Description

Displays a standard OLE 2.0 dialog allowing the user to choose whether to embed or link the OLE object on the clipboard when pasting it in the specified control. Embedding is the equivalent of calling the Paste function, and linking is the same as calling PasteLink.

Applies to

OLE 2.0 controls

Syntax

ole2control.**PasteSpecial** ( )

Parameter	Description
ole2control	The name of the OLE 2.0 control into which you want to paste the object on the clipboard

Return value

Integer. Returns *0* if it succeeds and one of the following values if an error occurs:

- *1* User canceled without selecting a paste option
- *-1* No data found
- *-9* Other error

Usage

See PasteLink for information about when an object on the clipboard is linkable.

Examples

If the clipboard contains an OLE object and the object's server application is running, then the following example lets the user choose to embed or link the object in the control ole_1:

```
integer result
result = ole_1.PasteSpecial( )
```

See also

LinkTo
Paste
PasteLink

Pi

Description

Multiplies pi by a specified number.

Syntax

Pi (*n*)

Parameter	Description
n	The number you want to multiply by pi (3.14159265358979323...)

Return value

Double. Returns the result of multiplying *n* by pi if it succeeds and *-1* if an error occurs.

Usage

Use Pi to convert angles to and from radians.

Examples

This statement returns pi:

```
Pi(1)
```

Both these statements return the area of a circle with the radius id_Rad, an instance variable of type double:

```
Pi(1) * id_Rad^2
```

```
Pi(id_Rad^2)
```

The following statements compute the cosine of a 45-degree angle:

```
real degree = 45.0, cosine
cosine = Cos(degree * (Pi(2)/360))
```

See also

Cos
Sin
Tan
Pi in Chapter 2, "DataWindow Painter Functions"

PixelsToUnits

Description

Converts pixels to PowerBuilder units. Because pixels are not usually square you also specify whether you are converting the pixels' horizontal or vertical measurement.

Syntax

PixelsToUnits (*pixels*, *type*)

Parameter	Description
pixels	An integer whose value is the number of pixels you want to convert to PowerBuilder units
type	A value of the ConvertType enumerated data type value indicating how to convert the value: ♦ XPixelsToUnits! — Convert the pixels in the horizontal direction ♦ YPixelsToUnits! — Convert the pixels in the vertical direction

Return value

Integer. Returns the converted value if it succeeds and *-1* if an error occurs.

Example

These statements convert 35 horizontal pixels to PowerBuilder units and set the variable Value equal to the converted value:

```
integer Value
Value = PixelsToUnits(35, XPixelsToUnits!)
```

See also

UnitsToPixels

PointerX

Description

Determines the distance of the pointer from the left edge of the specified object.

Applies to

Any object or control

Syntax

objectname.**PointerX** ( )

Parameter	Description
objectname	The name of the control or window for which you want the pointer's distance from the left edge. If you don't specify *objectname*, PointerX reports the distance from the left edge of the current sheet or window.

Return value

Integer. Returns the pointer's distance from the left edge of *objectname* in PowerBuilder units if it succeeds and *-1* if an error occurs.

Examples

In a script for a control in a window, the following example stores the distance of the pointer from the edge of the window in the variable li_dist. If the pointer is 5 units from the left edge of the window, li_dist equals 5:

```
integer li_dist
li_dist = Parent.PointerX( )
```

This statement in a control's RButtonDown script displays a popup menu m_Appl.M_Help at the cursor position:

```
m_Appl.m_Help.PopMenu(Parent.PointerX( ),  &
   Parent.PointerY( ))
```

If the previous example was part of the window's RButtonDown script, instead of a control in the window, the following statement displays the popup menu at the cursor position:

```
m_Appl.m_Help.PopMenu(This.PointerX( ),  &
   This.PointerY( ))
```

See also

PointerY
PopMenu
WorkSpaceHeight
WorkSpaceWidth
WorkSpaceX
WorkSpaceY

PointerY

Description

Determines the distance of the pointer from the top of the specified object.

Applies to

Any object or control

Syntax

objectname.**PointerY** ( )

Parameter	Description
objectname	The name of the control or window for which you want the pointer's distance from the top. If you don't specify *objectname*, PointerY reports the distance from the top of the current sheet or window.

Return value

Integer. Returns the pointer's distance from the top of *objectname* in PowerBuilder units if it succeeds and *-1* if an error occurs.

Examples

In a script for a control in a window, the following example stores the distance of the pointer from the top of the window in the variable li_dist. If the pointer is 10 units from the top of the window, li_dist equals 10:

```
integer li_Dist
li_Dist = Parent.PointerY( )
```

This statement in a control's RButtonDown script displays a popup menu m_Appl.M_Help at the cursor position:

```
m_Appl.M_Help.PopMenu(Parent.PointerX( ),  &
   Parent.PointerY( ))
```

See also

PointerX
PopMenu
WorkSpaceHeight
WorkSpaceWidth
WorkSpaceX
WorkSpaceY

PopMenu

Description

Displays a menu at the specified location.

Applies to

MenuItems in Menu objects

Syntax

menuitem.**PopMenu** (*xlocation*, *ylocation*)

Parameter	Description
menuitem	The fully qualified name of a MenuItem on a menu bar you want to display at the specified location
xlocation	The distance in PowerBuilder units of the displayed menu from the left edge of the window
ylocation	The distance in PowerBuilder units of the displayed menu from the top of the window

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If the menu object is not associated with the window so that it was opened when the window was opened, you must use CREATE to allocated memory for the menu (see the last example).

If the Visible attribute of the MenuItem is FALSE, you must make the MenuItem visible before you can display it as a popup menu.

The coordinates you specify for PopMenu are relative to the active window. In an MDI application, the coordinates are relative to the frame window, which is the active window. To display a menu at the cursor position, call PointerX and PointerY for the active window (the frame window in an MDI application) to get the coordinates of the cursor. (See the examples.)

Examples

These statements display the MenuItem m_Emp.M_Procedures at location 100, 200 in the active window. M_Emp is the menu associated with the window:

```
m_Emp.M_Procedures.PopMenu(100, 200)
```

This statement displays the MenuItem m_Appl.M_File at the cursor position, where m_Appl is the menu associated with the window:

```
m_Appl.M_file.PopMenu(PointerX( ), PointerY( ))
```

These statements display a popup menu at the cursor position. Menu4 was created in the Menu painter and has a MenuItem called m_language. Menu4 is not the menu for the active window. NewMenu is an instance of Menu4 (data type Menu4):

```
Menu4 NewMenu
NewMenu = CREATE Menu4
NewMenu.m_language.PopMenu(PointerX(), PointerY())
```

In an MDI application, the last line would include the MDI frame as the object for the pointer functions:

```
NewMenu.m_language.PopMenu(  &
   w_frame.PointerX(), w_frame.PointerY())
```

Pos

Description

Finds one string within another string.

Syntax

Pos (*string1*, *string2* {, *start* })

Parameter	Description
string1	The string in which you want to find *string2*.
string2	The string you want to find in *string1*.
start (optional)	A long indicating where the search will begin in *string1*. The default is 1.

Return value

Long. Returns a long whose value is the starting position of the first occurrence of *string2* in *string1* after the position specified in *start*. If *string2* is not found in *string1* or if *start* is not within *string1*, Pos returns *0*.

Usage

The Pos function is case sensitive.

Examples

This statement returns 6:

```
Pos("BABE RUTH", "RU")
```

This statement returns 1:

```
Pos("BABE RUTH", "B")
```

This statement returns 0:

```
Pos("BABE RUTH", "be") // The case does not match.
```

This statement starts searching at position 5 and returns 0, because position 5 is after the occurrence of BE:

```
Pos("BABE RUTH", "BE", 5)
```

These statements change the text NY in the SingleLineEdit sle_group to North East:

```
long place_nbr
place_nbr = Pos(sle_group.Text, "NY")
sle_group.SelectText(place_nbr, 2)
sle_group.ReplaceText("North East")
```

These statements separate the return value of GetBandAtPointer into the band name and row number. The Pos function finds the position of the tab in the string and the Left and Mid functions extract the information to the left and right of the tab.

```
string s, ls_left, ls_right
integer li_tab

s = dw_groups.GetBandAtPointer()
li_tab = Pos(s, "~t", 1)

ls_left = Left(s, li_tab - 1)
ls_right = Mid(s, li_tab + 1)
```

You could write similar code for a generic parsing function with three arguments. The string s would be an argument passed by value and ls_left and ls_right would be strings passed by reference.

Other functions that return a pair of tab-separated values for which you could use the parsing function are GetObjectAtPointer and GetValue.

See also

Left
Mid
Right
Pos in Chapter 2, "DataWindow Painter Functions"

Position

Description

Determines the position of the insertion point in an edit control.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox controls

Syntax

editname.**Position** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox in which you want to find the location of the cursor

Return value

Integer. Returns the location of the cursor in *editname* if it succeeds and *-1* if an error occurs.

Usage

Position reports the position number of the character immediately following the insertion point. For example, Position returns 1 if the cursor is at the beginning of *editname*. If text is selected in *editname*, Position reports the number of the first character of the selected text.

In a DataWindow control, Position reports the insertion point's position in the edit control over the current row and column.

Examples

If mle_EmpAddress contains Boston Street, the cursor is immediately after the n in Boston, and no text is selected, this statement returns 7:

```
mle_EmpAddress.Position( )
```

If mle_EmpAddress contains Boston Street and Street is selected, this statement returns 8 (the position of the S in Street):

```
mle_EmpAddress.Position( )
```

See also

SelectedLine
SelectedStart

Post

Description

Adds a message to the message queue for a window, either a PowerBuilder window or window of another application.

Syntax

Post (*handle, message#, word, long*)

Parameter	Description
handle	A long whose value is the system handle of a window (that you have created in PowerBuilder or another application) to which you want to post a message.
message#	An UnsignedInteger whose value is the system message number of the message you want to post.
word	A long whose value is the integer value of the message. If this argument is not used by the message, enter 0.
long	The long value of the message or a string.

Return value

Boolean.

Usage

Use Post or Send when you want to trigger system events that are not PowerBuilder-defined events. Post is asynchronous; it adds a message to the end of the window's message queue. Send is synchronous; its message triggers an event immediately.

To obtain the handle of a PowerBuilder window, use the Handle function.

To trigger PowerBuilder events, use TriggerEvent or PostEvent. These functions run the script associated with the event. They are easier to code and bypass the messaging queue.

When you specify a string for *long*, Post stores a copy of the string and passes a pointer to it.

Platform information
On the Macintosh, you can use Handle to get the handle of a PowerBuilder object and post a message to it. However, you cannot use Handle to get the handle of external objects.

Example

This statement scrolls the window w_date down one page after all the previous messages in the message queue for the window have been processed:

```
Post(Handle(w_date), 277, 3, 0)
```

See also

Handle
PostEvent
Send
TriggerEvent

PostEvent

Description

Adds an event to the end of the event queue of an object.

Applies to

Any object, except the application object

Syntax

objectname.**PostEvent** (*event*, { *word*, *long* })

Parameter	Description
objectname	The name of any PowerBuilder object or control (except an application) that has events associated with it.
event	A value of the TrigEvent enumerated data type that identifies a PowerBuilder event (for example, Clicked!, Modified!, or DoubleClicked!) or a string whose value is the name of an event. The event must be a valid event for *objectname* and a script must exist for the event in *objectname*.
word (optional)	A long value to be stored in the WordParm attribute of the system's Message object. If you want to specify a value for *long*, but not *word*, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs.)
long (optional)	A long value or a string that you want to store in the LongParm attribute of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm attribute, which you can access with the String function (see Usage).

Return value

Boolean. Returns TRUE if it is successful and FALSE if the event is not a valid event for *objectname* or no script exists for the event in *objectname*.

Usage

You cannot post events to the event queue for an application object. Use TriggerEvent instead.

You cannot post or trigger events for objects that don't have events, such as drawing objects. You cannot post or trigger events in a batch application that has no user interface because the application has no event queue.

After you call PostEvent, check the return code to determine whether PostEvent succeeded.

You can pass information to the event script with the *word* and *long* arguments. The information is stored in the Message object. In your script, you can reference the WordParm and LongParm fields of the Message object to access the information. Note that the Message object is saved and restored just before the posted event script runs so that the information you passed is available even if other code has used the Message object too.

If you have specified a string for *long*, you can access it in the triggered event by using the String function with the keyword "address" as the *format* parameter. (Note that PowerBuilder has stored the string at an arbitrary memory location and you are relying on nothing else having altered the pointer or the stored string.) Your event script might begin as follows:

```
string PassedString
PassedString = String(Message.LongParm, "address")
```

TriggerEvent and PostEvent are useful for preventing duplication of code. If two controls perform the same task, you can use PostEvent in one control's event script to execute the other's script, instead of repeating the code in two places. For example, if both a button and a MenuItem delete data, the button's Clicked script can perform the deletion and the MenuItem's Clicked event script can post an event that runs the button's Clicked event script.

Choosing PostEvent or TriggerEvent

Both PostEvent and TriggerEvent cause event scripts to be executed. PostEvent is asynchronous; it adds the event to the end of an object's event queue. TriggerEvent is synchronous; the event is triggered immediately.

Use PostEvent when you want the current event script to complete before the posted event script runs. TriggerEvent interrupts the current script to run the triggered event's script. Use it when you need to interrupt a process, such as canceling printing.

If the function is the last line in an event script and there are no other events pending, PostEvent and TriggerEvent have the same effect.

Events and messages in Windows

Both PostEvent and TriggerEvent cause a script associated with an event to be executed. However, these functions do not send the actual event message. This is important when you are choosing the target object and event. The following background information explains this concept.

Many PowerBuilder functions send Windows messages, which in turn trigger events and run scripts. For example, the Close function sends a Windows close message (WM_CLOSE), which PowerBuilder maps to its internal close message (PBM_CLOSE). Then it runs the Close event's script and closes the window.

If you use TriggerEvent or PostEvent with Close! as the argument, PowerBuilder runs the Close event's script but it does *not* close the window because it didn't receive the close message. Therefore, the choice of which event to trigger is important. If you trigger the Clicked! event for a button whose script calls the Close function, then PowerBuilder runs the Close event's script *and* closes the window.

Use Post or Send when you want to trigger system events that are not PowerBuilder-defined events.

Examples

This statement adds the Clicked event to the event queue for CommandButton cb_OK. The event script will be executed after any other pending event scripts are run:

```
cb_OK.PostEvent(Clicked!)
```

This statement adds the user-defined event cb_exit_request to the event queue in the parent window:

```
Parent.PostEvent("cb_exit_request")
```

This example posts an event for cb_exit_request with an argument and then retrieves that value from the Message object in the event's script.

The first part of the example is code for a button in a window. It adds the user-defined event cb_exit_request to the event queue in the parent window. The value 455 is stored in the Message object for the use of the event's script:

```
Parent.PostEvent("cb_exit_request", 455)
```

The second part of the example is the beginning of the cb_exit_request event script, which assigns the value passed in the Message object to a local variable. The script can use the value in whatever way is appropriate to the situation:

```
integer numarg
numarg = Message.WordParm
```

See also

Post
Send
TriggerEvent

Print

Description

Sends data to the current printer (or spooler, if the user has a spooler set up). There are several syntaxes:

- To send the contents of a DataWindow control to the printer as a print job, use Syntax 1.
- To include a visual object, such as a window or a graph control in a print job, use Syntax 2.
- To send one or more lines of text as part of a print job, use Syntax 3. The text can be preceded or followed by tab settings, which control the text's horizontal position on the page.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) Any object

(Syntax 3) Not object-specific

Syntax 1

datawindowname.**Print** ({ *canceldialog* })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that contains the information to be printed
canceldialog (optional)	A boolean value indicating whether you want to display a non-modal dialog that allows the user to cancel printing. Values are: ♦ TRUE — (Default) Display the dialog ♦ FALSE — Do not display the dialog

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

objectname.**Print** (*printjobnumber*, *x*, *y* {, *width*, *height* })

Parameter	Description
objectname	The name of the object that you want to print. The object must either be a window or an object whose ancestor type is DragObject, which includes all the controls that you can place in a window.
printjobnumber	The number the PrintOpen function assigns to the print job.
x	An integer whose value is the x coordinate on the page of the left corner of the object, in thousandths of an inch.
y	An integer whose value is the y coordinate on the of the left corner of the object page, in thousandths of an inch.
width (optional)	An integer specifying the printed width of the object in thousandths of an inch. If omitted, PowerBuilder uses the object's original width.
height (optional)	An integer specifying the printed height of the object in thousandths of an inch. If omitted, PowerBuilder uses the object's original height.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 3

Print (*printjobnumber*, { *tab1*, } *string* {, *tab2* })

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job.
tab1 (optional)	The position, measured from the left edge of the print area in thousandths of a inch, to which the print cursor should move before *string* is printed. If the print cursor is already at or beyond the position or if you omit *tab1*, Print starts printing at the current position of the print cursor.
string	The string you want to print. If the string includes carriage return-newline character pairs (~r~n), the string will print on multiple lines. However, the initial tab position is ignored on subsequent lines.
tab2 (optional)	The new position, measured from the left edge of the print area in thousandths of a inch, of the print cursor after *string* printed. If the print cursor is already at or beyond the specified position, Print ignores *tab2* and the print cursor remains at the end of the text. If you omit *tab2*, Print moves the print cursor to the beginning of a new line.

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

PowerBuilder manages print jobs by opening the job, sending data, and closing the job. When you use Syntax 1, print job management happens automatically. When you use Syntax 2 and 3, you must call the PrintOpen function and the PrintClose or PrintCancel functions yourself to manage the process.

Use Syntax 1 to print the contents of a DataWindow object. The Print function prints all the rows that have been retrieved. To print several DataWindows as a single job, don't use Print. Instead, open the print job with PrintOpen, call the PrintDataWindow function for each DataWindow, and close the job.

Events for DataWindow printing
When you use Print for DataWindow controls, it triggers a PrintStart event just before any data is sent to the printer (or spooler), a PrintPage event for each page break, and a PrintEnd event when printing is complete.

The PrintPage event has action codes that let you control whether the page about to be formatted is printed. You can skip the upcoming page by calling SetActionCode in the PrintPage event and setting the action code to 1.

Print cursor
In a print job, PowerBuilder uses a print cursor to keep track of the print location. The print cursor stores the coordinates of the upper-left corner of the location at which print will being. PowerBuilder updates the print cursor after printing text with Print.

Line spacing when printing text
Line spacing in PowerBuilder is proportional to character height. The default line spacing is 1.2 times the character height. When Print starts a new line, it sets the x coordinate of the cursor to 0 and increases the y coordinate by the current line spacing. You can change the line spacing with the PrintSetSpacing function, which lets you specify a new factor to be multiplied by the character height.

Because Syntax 3 of Print increments the y coordinate each time it creates a new line, it also handles page breaks automatically. When the y coordinate exceeds the page size, PowerBuilder automatically creates a new page in the print job. You don't need to call the PrintPage function, as you would if you were using the printing functions that control the cursor position (for example, PrintText or PrintLine).

Print area and margins
The print area is the physical page size minus any margins in the printer itself. Depending on the printer, you may be able to change margins using PrintSend and printer-defined escape sequences.

Using fonts
You can use PrintDefineFont and PrintSetFont to specify the font used by the Print function when you are printing a string. When you use Syntax 1 to print a DataWindow, PowerBuilder uses the fonts and layout that appear in the DataWindow.

Examples

This statement sends the contents of dw_employee to the current printer:

```
dw_employee.Print( )
```

Syntax 2

This example prints the CommandButton cb_close in its original size at location 500, 1000:

```
integer Job
Job = PrintOpen( )
cb_close.Print(Job, 500,1000)
PrintClose(Job)
```

Syntax 2 and 3

This example opens a print job, which defines a new page, then prints a title and a graph on the first page and a window on the second page:

```
integer Job
Job = PrintOpen( )
Print(Job, "Report of Year-to-Date Sales")
gr_sales1.Print(Job, 1000,PrintY(Job)+500,  &
   6000,4500)
PrintPage(Job)
w_sales.Print(Job, 1000,500, 6000,4500)
PrintClose(Job)
```

Syntax 3

This example opens a print job, prints the string Powersoft Corporation in the default font, and then starts a new line:

```
integer Job

// Define a blank page and assign the job an ID
Job = PrintOpen( )

// Print the string and then start a new line
Print(Job, "Powersoft Corporation")
. . .
PrintClose(Job)
```

This example opens a print job, prints the string Powersoft Corporation in the default font, tabs 5 inches from the left edge of the print area but does not start a new line:

```
integer Job

// Define a blank page and assign the job an ID
Job = PrintOpen( )

// Print the string but do not start a new line
Print(Job, "Powersoft Corporation", 5000)
. . .
PrintClose(Job)
```

The first Print statement below tabs half an inch from the left edge of the print area, prints the string Powersoft Corporation, and then starts a new line. The second Print statement tabs one inch from the left edge of the print area, prints the string Directors:, and then starts a new line:

```
integer Job

// Define a blank page and assign the job an ID
Job = PrintOpen( )

// Print the string and start a new line
Print(Job, 500, "Powersoft Corporation")

// Tab 1 inch from the left edge and print
Print(Job, 1000, "Directors:")
. . .
PrintClose(Job)
```

The first Print statement below tabs half an inch from the left edge of the print area prints the string Powersoft Corporation, and then tabs 6 inches from the left edge of the print area but does not start a new line. The second Print statement prints the current date and then starts a new line:

```
integer Job

// Define a blank page and assign the job an ID
Job = PrintOpen( )

// Print string and tab 6 inches from the left edge
Print(Job, 500, "Powersoft Corporation", 6000)

// Print the current date on the same line
Print(Job, String(Today()))
. . .
PrintClose(Job)
```

In a window that displays a database error message in a MultiLineEdit mle_message, the following script for a Print button prints a title with the date and time and the message:

```
integer li_prt

li_prt   = PrintOpen("Database Error")

Print(li_prt, "Database error  - "  &
   + String(Today(), "mm/dd/yyyy")  &
   + " - "  &
   + String(Now(), "HH:MM:SS"))
Print(li_prt, " ")
Print(li_prt, mle_message.text)

PrintClose(li_prt)
```

See also

PrintCancel
PrintClose
PrintDataWindow
PrintOpen
PrintScreen
PrintSetFont
PrintSetSpacing

PrintBitmap

Description

Writes a bitmap at the specified location on the current page.

Syntax

PrintBitmap (*printjobnumber, bitmap, x, y, width, height*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
bitmap	A string whose value is the file name of the bitmap image
x	An integer whose value is the x coordinate (in thousandths of an inch) on the page of the bitmap image
y	An integer whose value is the y coordinate (in thousandths of an inch) on the page of the bitmap image
width	The integer width of the bitmap image in thousandths of an inch. If *width* is 0, PowerBuilder uses the original width of the image.

Parameter	Description
height	The integer height of the bitmap image in thousandths of an inch. If *height* is 0, PowerBuilder uses the original height of the image.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

PrintBitmap does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in See also).

Example

These statements define a new blank page and then print the bitmap in file d:\PB\BITMAP1.BMP in its original size at location 50,100:

```
integer Job

// Define a new blank page.
Job = PrintOpen( )

// Print the bitmap in its original size.
PrintBitmap(Job, "d:\PB\BITMAP1.BMP", 50,100, 0,0)

// Send the page to the printer and close Job.
PrintClose(Job)
```

See also

PrintClose
PrintLine
PrintRect
PrintRoundRect
PrintOval
PrintOpen

PrintCancel

Description

Cancels printing and deletes the spool file, if any. There are two syntaxes:

- Syntax 1 cancels printing of a DataWindow printed with the Print function

- Syntax 2 cancels a print job that you began with the PrintOpen function.

Applies to

Syntax 1: DataWindow controls and child DataWindows

Syntax 2: Not object-specific

Syntax 1

datawindowname.**PrintCancel** ( )

Parameter	Description
datawindowname	The name of a DataWindow control or child DataWindow that was printed with Syntax 1 of the Print function

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

PrintCancel (*printjobnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

PrintCancel cancels the specified print job or the printing of the specified DataWindow by deleting the spool file, if any, and closing the job. Because PrintCancel closes the print job, do not call the PrintClose function after you call PrintCancel.

When you use the Print function to print the DataWindow, without using PrintOpen, use Syntax 1 to cancel printing. When you use the PrintDataWindow function to print a DataWindow as part of a print job, use Syntax 2 to cancel printing.

Examples

These statements send the contents of the DataWindow dw_employee to the current printer without displaying the Cancel dialog and then cancel the printing:

```
dw_Employee.Print(FALSE)
dw_employee.PrintCancel( )
```

Syntax 2

In this example, a script for a Print button opens a print job and then opens a window with a cancel button. If the user clicks on the cancel button, its script sets a global variable that indicates that the user wants to cancel the job. After each printing command in the Print button's script, the code checks the global variable and cancels the job if its value is TRUE.

The definition of the global variable is:

```
boolean gb_printcancel
```

The script for the Print button is:

```
integer job, li

gb_printcancel = FALSE
job = PrintOpen("Test Page Breaks")
IF job < 1 THEN
   MessageBox("Error", "Can't open a print job.")
   RETURN
END IF

Open(w_printcancel)

PrintBitmap(Job, "d:\PB\bitmap1.bmp", 5, 10, 0, 0)
IF gb_printcancel = TRUE THEN
   PrintCancel(job)
   RETURN
END IF

...  // Additional printing commands,
...  // including checking gb_printcancel

PrintClose(job)
Close(w_printcancel)
```

The script for the cancel button in the second window is:

```
gb_printcancel = TRUE
Close(w_printcancel)
```

See also

Print
PrintClose
PrintOpen

PrintClose

Description

Sends the current page to the printer (or spooler) and closes the job. Call PrintClose as the last command of a print job unless PrintCancel function has closed the job.

Syntax

PrintClose (*printjobnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

When you open a print job, you must close (or cancel) it. To avoid hung print jobs, process and close a print job in the same event in which you open it.

Example

This example opens a print job, which creates a blank page, prints a bitmap on the page, then sends the current page to the printer or spooler and closes the job:

```
integer Job

// Begin a new job and a new page.
Job = PrintOpen( )

// Print the bitmap in its original size.
PrintBitmap(Job, d:\PB\BITMAP1, 5,10, 0,0)

// Send the page to the printer and close Job.
PrintClose(Job)
```

See also

PrintCancel
PrintOpen

PrintDataWindow

Description

Prints the contents of a DataWindow control as a print job.

Syntax

PrintDataWindow (*printjobnumber, datawindowname*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
datawindowname	The name of the DataWindow control or child DataWindow to be printed

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Do not use PrintDataWindow with any Print functions except PrintOpen and PrintClose.

When you use PrintDataWindow with PrintOpen and PrintClose, you can print several DataWindows in one print job. The information in each DataWindow control starts printing on a new page.

When you print a DataWindow, PowerBuilder uses the fonts and layout that appear in the DataWindow. PrintDefineFont and PrintSetFont have no effect.

See the Print function for information on skipping individual pages by setting an action code in the PrintPage event.

Example

These statements send the contents of three DataWindow controls to the current printer in a single print job:

```
integer job
job = PrintOpen( )
// Each DataWindow starts printing on a new page.
PrintDataWindow(job, dw_EmpHeader)
PrintDataWindow(job, dw_EmpDetail)
PrintDataWindow(job, dw_EmpDptSum)
PrintClose(job)
```

See also

Print
PrintClose
PrintOpen

PrintDefineFont

Description

Creates a numbered font definition that consists of a font supported by your printer and a set of font attributes. You can use the font number in the PrintSetFont or PrintText functions. You can define up to 8 fonts at a time.

Syntax

PrintDefineFont (*printjobnumber, fontnumber, facename, & height, weight, fontpitch, fontfamily, italic, underline*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job.
fontnumber	The number (1 to 8) you want to assign to the font.
facename	A string whose value is the name of a typeface supported by your printer (for example, Courier 10Cpi).
height	The height of the type in thousandths of an inch (for example, 250 for 18-point 10Cpi) or a negative number representing the point size (for example, -18 for 18 point). Specifying the point size is more exact; the height in thousandths of an inch only approximates the point size.
weight	The stroke weight of the type. Normal weight is 400 and bold is 700.
fontpitch	A value of the FontPitch enumerated data type indicating the pitch of the font: ♦ Default! ♦ Fixed! ♦ Variable!
fontfamily	A value of the FontFamily enumerated data type indicating the family of the font: ♦ AnyFont! ♦ Decorative! ♦ Modern! ♦ Roman! ♦ Script! ♦ Swiss!
italic	A boolean value indicating whether the font is italic. The default is FALSE, that is, not italic.

Parameter	Description
underline	A boolean value indicating whether the font is underlined. The default is FALSE, that is, not underlined.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

You can use as many as eight fonts in one print job. If you require more than eight fonts in one job, you can call PrintDefineFont again to change the settings for a font number.

Use PrintSetFont to make a font number the current font for the open print job.

> **Fonts in Microsoft Windows**
> Although the *fontfamily* argument seems to duplicate information in the font name, Windows uses it along with the font name to identify the correct font or substitute a similar font if the named font is unavailable.
>
> **Fonts on the Macintosh**
> The Macintosh uses the *facename*, *height*, and *weight* arguments to select the font. *Fontfamily* and *fontpitch* are ignored.

> **Font names and sizes**
> Some font names include a size, especially monospaced fonts which include characters per inch. This is the recommended size for the font and doesn't affect the printed size, which you specify with the *height* argument.

Example

These statements define a new blank page, and then define print font 1 for Job as Courier 10Cpi, 18 point, normal weight, default pitch, Decorative font, with no italic or underline:

```
integer Job
Job = PrintOpen( )
PrintDefineFont(Job, 1, "Courier 10Cpi",  &
   -18, 400, Default!, Decorative!, FALSE, FALSE)
```

See also

PrintClose
PrintOpen
PrintSetFont

PrintLine

Description

Draws a line of a specified thickness between the specified endpoints on the current print page.

Syntax

PrintLine (*printjobnumber, x1, y1, x2, y2, thickness*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
x1	An integer specifying the x coordinate in thousandths of an inch of the start of the line
y1	An integer specifying the y coordinate in thousandths of an inch of the start of the line
x2	An integer specifying the x coordinate in thousandths of an inch of the end of the line
y2	An integer specifying the y coordinate in thousandths of an inch of the end of the line
thickness	An integer specifying the thickness of the line in thousandths of an inch

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

PrintLine does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in Related topics below).

Example

These statements start a new page in a print job and then print a line starting at 0,5 and ending at 7500,5 with a thickness of 10/1000 of an inch:

```
integer Job

// Open a job.
Job = PrintOpen( )
... // various print commands

// Start a new page.
PrintPage(Job)

// Print a line at the top of the page,
// starting a 0,5 and ending at 7500,5.
PrintLine(Job,0,5,7500,5,10)
...  // Other printing
PrintClose(Job)
```

See also

PrintBitmap
PrintClose
PrintOpen
PrintOval
PrintRect
PrintRoundRect

PrintOpen

Description

Opens a print job and assigns it a number, which you use in other printing statements.

Syntax

PrintOpen({ *jobname* })

Parameter	Description
jobname (optional)	A string specifying a name for the print job. The name is displayed in the Windows 3.x Print Manager dialog box and in the Spooler dialog box.

Return value

Integer. Returns the job number if it succeeds and *-1* if an error occurs.

Usage

A new print job begins on a new page and the font is set to the default font for the printer. The print cursor is at the upper left corner of the print area.

Use the job number that PrintOpen returns to identify this print job in all subsequent print functions.

Calling MessageBox after PrintOpen can cause undesirable behavior that is confusing to a user. Calling PrintOpen causes the currently active window in PowerBuilder to be disabled to allow Windows to handle printing. If you display a MessageBox after calling PrintOpen, Windows assigns the active window to be its parent, which is often another application, causing that application to become active.

Balancing PrintOpen and PrintClose
When you open a print job, you must close (or cancel) it. To avoid hung print jobs, process and close a print job in the same event in which you open it.

Examples

This example opens a job but does not give it a name:

```
integer li_job
li_job = PrintOpen( )
```

This example opens a job and gives it a name:

```
integer li_job
li_job = PrintOpen("Phone List")
```

See also

Print
PrintBitmap
PrintCancel
PrintClose
PrintDataWindow
PrintDefineFont
PrintLine
PrintOval
PrintPage
PrintRect
PrintRoundRect
PrintSend
PrintSetFont
PrintSetup
PrintText
PrintWidth
PrintX
PrintY

PrintOval

Description

Draws a white oval outlined in a line of the specified thickness on the print page.

Syntax

PrintOval (*printjobnumber, x, y, width, height, thickness*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
x	An integer specifying the x coordinate in thousandths of an inch of the upper left corner of the oval's bounding box
y	An integer specifying the y coordinate in thousandths of an inch of the upper left corner of the oval's bounding box
width	An integer specifying the width in thousandths of an inch of the oval's bounding box
height	An integer specifying the height in thousandths of an inch of the oval's bounding box
thickness	An integer specifying the thickness of the line that outlines the oval in thousandths of an inch

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The PrintOval, PrintRect, and PrintRoundRect functions draw filled shapes. To print other shapes or text inside the shapes, draw the filled shape first and then add text and other shapes or lines inside it. If you draw the filled shape after other printing functions, it will cover anything inside it. For example, to draw a border around text and lines, draw the oval or rectangular border first and then use PrintLine and PrintText to position the lines and text inside.

PrintOval does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in See also).

Example

This example starts a print job with a new blank page, and then prints an oval that fits in a 1-inch square. The upper left corner of the oval's bounding box is 4 inches from the top and 3 inches from the left edge of the print area. Because its height and width are equal, the oval is actually a circle:

```
integer Job
// Define a new blank page.
Job = PrintOpen( )
// Print an oval.
PrintOval(Job, 4000, 3000, 1000, 1000, 10)
...  // Other printing
PrintClose(Job)
```

See also

PrintBitmap
PrintClose
PrintLine
PrintOpen
PrintRect
PrintRoundRect

PrintPage

Description

Sends the current page to the printer or spooler and begins a new blank page in the current print job.

Syntax

PrintPage (*printjobnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

This example opens a print job with a new blank page, prints a bitmap on the page, and then sends the page to the printer and sets up a new blank page. Finally, the last Print statement prints the company name on the new page:

```
integer Job

// Open a job with new blank page.
Job = PrintOpen( )

// Print a bitmap on the page.
PrintBitmap(Job, d:\PB\BITMAP1, 100,250, 0,0)

// Begin a new page.
PrintPage(Job)

// Print the company name on the new page.
Print(Job, "Powersoft Corporation")
```

See also

PrintClose
PrintOpen

PrintRect

Description

Draws a white rectangle with a border of the specified thickness on the print page.

Syntax

PrintRect (*printjobnumber, x, y, width, height, thickness*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
x	An integer specifying the x coordinate in thousandths of an inch of the upper left corner of the rectangle
y	An integer specifying the y coordinate in thousandths of an inch of the upper left corner of the rectangle
width	An integer specifying the rectangle's width in thousandths of an inch
height	An integer specifying the rectangle's height in thousandths of an inch
thickness	An integer specifying the thickness of the rectangle's border line in thousandths of an inch

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The PrintOval, PrintRect, and PrintRoundRect functions draw filled shapes. To print other shapes or text inside the shapes, draw the filled shape first and then add text and other shapes or lines inside it. If you draw the filled shape after other printing functions, it will cover anything inside it. For example, to draw a border around text and lines, draw the oval or rectangular border first and then use PrintLine and PrintText to position the lines and text inside.

PrintRect does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in See also below).

Example

These statements open a print job with a new page and draw a 1-inch square with a line thickness of 1/8 of an inch. The square's upper left corner is 4 inches from the left and 3 inches from the top of the print area:

```
integer Job
// Define a new blank page.
Job = PrintOpen( )
// Print the rectangle on the page.
PrintRect(Job, 4000,3000, 1000,1000, 125)
...  // Other printing
PrintClose(Job)
```

See also

PrintBitmap
PrintClose
PrintLine
PrintOpen
PrintOval
PrintRoundRect

PrintRoundRect

Description

Draws a white rectangle with rounded corners and a border of the specified thickness on the print page.

Syntax

PrintRoundRect (*printjobnumber, x, y, width, height, xradius, & yradius, thickness*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
x	An integer specifying the x coordinate in thousandths of an inch of the upper left corner of the rectangle
y	An integer specifying the y coordinate in thousandths of an inch of the upper left corner of the rectangle
width	An integer specifying the rectangle's width in thousandths of an inch
height	An integer specifying the rectangle's height in thousandths of an inch
xradius	An integer specifying the x radius of the corner rounding
yradius	An integer specifying the y radius of the corner rounding
thickness	An integer specifying the thickness of the rectangle's border line in thousandths of an inch

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The PrintOval, PrintRect, and PrintRoundRect functions draw filled shapes. To print other shapes or text inside the shapes, draw the filled shape first and then add text and other shapes or lines inside it. If you draw the filled shape after other printing functions, it will cover anything inside it. For example, to draw a border around text and lines, draw the oval or rectangular border first and then use PrintLine and PrintText to position the lines and text inside.

PrintRoundRect does not change the position of the print cursor, which remains where it was before the function was called. In general, print functions in which you specify coordinates do not affect the print cursor (see the functions listed in See also).

Example

This example starts a new print job, which begins a new page, and prints a rectangle with rounded corners as a page border. Then it closes the print job, which sends the page to the printer. The rectangle is 6 1/4 inches wide by 9 inches high and its upper corner is 1 inch from the top and 1 inch from the left edge of the print area. The border has a line thickness of 1/8 of an inch and the corner radius is 300:

```
integer Job

// Define a new blank page.
Job = PrintOpen( )

// Print a RoundRectangle on the page.
PrintRoundRect(Job, 1000,1000, 6250,9000,  &
      300,300, 125)

// Send the page to the printer.
PrintClose(Job)
```

See also

PrintBitmap
PrintClose
PrintLine
PrintOpen
PrintOval
PrintRect

PrintScreen

Description

Prints the screen image as part of a print job.

Syntax

PrintScreen (*printjobnumber*, *x*, *y* {, *width* {, *height* } })

Parameter	Description
printjobnumber	The number the PrintOpen function assigns to the print job.
x	An integer whose value is the x coordinate on the page, in thousandths of an inch, of the upper left corner of the screen image.

Parameter	Description
y	An integer whose value is the y coordinate on the page, in thousandths of an inch, of the upper left corner of the screen image.
width (optional)	The integer width of the printed screen in thousandths of an inch. If you omit *width*, PowerBuilder prints the screen at its original width.
height (optional)	The integer height of the printed screen in thousandths of an inch. If you omit *height*, PowerBuilder prints the screen at its original height.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

This statement prints the current screen image in its original size at location 500, 1000:

```
integer Job
Job = PrintOpen( )
PrintScreen(Job, 500,1000)
PrintClose(Job)
```

See also

Print
PrintClose
PrintOpen

PrintSend

Description

Sends an arbitrary string of characters to the printer. PrintSend is usually used for sending escape sequences that change the printer's setup.

Syntax

PrintSend (*printjobnumber, string* {, *zerochar* })

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job.
string	A string you want to send to the printer. In the string, use ASCII values for nonprinting characters.

Parameter	Description
zerochar (optional)	An ASCII value (1 to 255) that you want to use to represent the number zero in *string*.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use PrintSend to send escape sequences to specific printers (for example, to set condensed mode or to set margins). Escape sequences are printer specific.

As with any string, the number zero terminates the *string* argument. If the printer code you want to send includes a 0, you can use another character for 0 in *string* and specify the character that represents 0 in *zerochar*. The character you select should be a character you don't usually use. When PowerBuilder sends the string to the printer it converts the substitute character to a 0.

A typical print job, in which you want to make printer-specific settings, might consist of the following function calls:

1 PrintOpen

2 PrintSend, to change the printer orientation, select a tray, etc.

3 PrintDefineFont and PrintSetFont to specify fonts for the job

4 Print to output job text

5 PrintClose

Examples

This example opens a print job and sends an escape sequence to a printer in IBM Proprinter mode to change the margins. There is no need to designate a character to represent 0:

```
integer Job

// Open a print job.
Job = PrintOpen( )

/* Send the escape sequence.
1B is the escape character in hexadecimal.
X indicates that you are changing the margins.
030 sets the left margin to 30 character spaces.
040 sets the right margin to 40 character spaces.
*/
```

```
PrintSend(Job,"~h1BX~030~040")
...  // Print text or DataWindow

// Send the job to the printer or spooler.
PrintClose(Job)
```

This example opens a print job and sends an escape sequence to a printer in IBM Proprinter mode to change the margins. The decimal ASCII code 255 represents zero:

```
integer Job

// Open a print job.
Job = PrintOpen( )

/* Send the escape sequence.
1B is the escape character, in hexadecimal.
X indicates that you are changing the margins.
255 sets the left margin to 0.
040 sets the right margin to 40 character spaces.
*/
PrintSend(Job, "~h1BX~255~040", 255)
PrintDataWindow(Job, dw_1)

// Send the job to the printer or spooler.
PrintClose(Job)
```

See also

PrintClose
PrintOpen

PrintSetFont

Description

Designates a font to be used for text printed with the Print function. You specify the font by number. Use PrintDefineFont to associate a font number with the desired font, a size, and a set of attributes.

Syntax

PrintSetFont (*printjobnumber, fontnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
fontnumber	The number (1 to 8) of a font defined for the job in PrintDefineFont or 0 (the default font for the printer)

Return value

Integer. Returns the character height of the current font if it succeeds and *-1* if an error occurs.

Example

This example starts a new print job and specifies that font number 2 is Courier, 18 point, bold, default pitch, in modern font, with no italic or underline. The PrintSetFont statement sets the current font to font 2. Then the Print statement prints the company name:

```
integer Job

// Start a new print job and a new page.
Job = PrintOpen( )

// Define the font for Job.
PrintDefineFont(Job, 2, "Courier 10Cps",  &
   250, 700, Default!, Modern!, FALSE, FALSE)

// Set the font for Job.
PrintSetFont(Job, 2)

// Print the company name in the specified font.
Print(Job,"Powersoft Corporation")
```

See also

PrintDefineFont
PrintOpen

PrintSetSpacing

Description

Sets the factor that PowerBuilder uses to calculate line spacing.

Syntax

PrintSetSpacing (*printjobnumber, spacingfactor*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job.
spacingfactor	The number by which you want to multiply the character height to determine the vertical line-to-line spacing. The default is 1.2.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Line spacing in PowerBuilder is proportional to character height. The default line spacing is 1.2 times the character height. When Print starts a new line, it sets the x coordinate of the cursor to 0 and increases the y coordinate by the current line spacing. The PrintSetSpacing function lets you specify a new factor to be multiplied by the character height for an open print job.

Example

These statements start a new print job and set the vertical spacing factor to 1.5 (one and a half spacing):

```
integer Job

// Define a new blank page.
Job = PrintOpen( )

// Set the spacing factor.
PrintSetSpacing(Job, 1.5)
```

See also

PrintOpen

PrintSetup

Description

Calls the Printer Setup dialog box provided by the system printer driver and stores the user's responses in the printer driver.

Syntax

PrintSetup ( )

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

These statements call the Printer Setup dialog box for the current system printer and then start a new print job:

```
integer Job

// Call the printer setup program.
PrintSetup( )

// Start a job and a new page.
Job = PrintOpen( )
```

See also

PrintOpen

PrintText

Description

Prints a single line of text starting at the specified coordinates.

Syntax

PrintText (*printjobnumber, string, x, y* {, *fontnumber*})

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job.
string	A string whose value is the text you want to print.
x	An integer specifying the x coordinate in thousandths of an inch of the beginning of the text.
y	An integer specifying the y coordinate in thousandths of an inch of the beginning of the text.
fontnumber (optional)	The number (1 to 8) of a font defined for the job by using the PrintDefineFont function or 0 (the default font for the printer). If you omit *fontnumber*, the text prints in the current font for the print job.

Return value

Integer. Returns the x coordinate of the new cursor location (that is, the value of the parameter *x* plus the width of the text) if it succeeds. PrintText returns *-1* if an error occurs.

Usage

PrintText does change the position of the print cursor, unlike the other print functions for which you specify coordinates. The print cursor moves to the end of the printed text. PrintText also returns the x coordinate of the print cursor. You can use the return value to determine where to begin printing additional text.

PrintText does not change the print cursor's y coordinate, which is its vertical position on the page.

Examples

These statements start a new print job and then print PowerBuilder in the current font 3.7 inches from the left edge at the top of the page (location 3700,10):

```
integer Job

// Define a new blank page.
Job = PrintOpen( )
```

```
// Print the text.
PrintText(Job,"PowerBuilder", 3700, 10)
...  // Other printing
PrintClose(Job)
```

The following statements define a new blank page and then print Confidential in bold (as defined for font number 3), centered at the top of the page:

```
integer Job

// Start a new job and a new page.
Job = PrintOpen( )

// Define the font.
PrintDefineFont(Job, 3, &
   "Courier 10Cps", 250,700,  &
      Default!, AnyFont!, FALSE, FALSE)

// Print the text.
PrintText(Job, "Confidential", 3700, 10, 3)
...  // Other printing
PrintClose(Job)
```

This example prints four lines of text in the middle of the page. The coordinates for PrintText establish a new vertical position for the print cursor, which the subsequent Print functions use and increment. The first Print function uses the x coordinate returned by PrintText to continue the first line. The rest of the Print functions print additional lines of text, after tabbing to the x coordinate used initially by PrintText. In this example, each Print function increments the y coordinate so that the following Print function starts a new line:

```
integer Job

// Start a new job and a new page.
Job = PrintOpen( )

// Print the text.
x = PrintText(Job,"The material ", 2000, 4000)
Print(Job, x, " in this report")
Print(Job, 2000, "is confidential and should not")
Print(Job, 2000, "be disclosed to anyone who")
Print(Job, 2000, "is not at this meeting.")
...  // Other printing
PrintClose(Job)
```

See also

Print
PrintClose
PrintOpen

PrintWidth

Description

Determines the width of a string using the current font of the specified print job.

Syntax

PrintWidth (*printjobnumber, string*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job
string	A string whose value is the text for which you want to determine the width

Return value

Integer. Returns the width of *string* in thousandths of an inch using the current font of *printjobnumber* if it succeeds and *-1* if an error occurs.

Example

These statements define a new blank page and then set W to the length of the string PowerBuilder in the current font and then use the length to position the next text line:

```
integer Job, W

// Start a new print job.
Job = PrintOpen( )

// Determine the width of the text.
W = PrintWidth(Job,"PowerBuilder")

// Use the width to get the next print position.
Print(Job, W - 500, "Features List")
```

See also

PrintClose
PrintOpen

PrintX

Description

Reports the x coordinate of the print cursor.

Syntax

PrintX (*printjobnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job

Return value

Integer. Returns the x coordinate of the print cursor if it succeeds and *-1* if an error occurs.

Example

These statements set LocX to the x coordinate of the cursor and print End of Report an inch beyond that location:

```
integer LocX, Job

Job = PrintOpen( )
. . .  //Print statements
LocX = PrintX(Job)
Print(LocX+1000, "End of Report")
```

See also

PrintY

PrintY

Description

Reports the y coordinate of the print cursor.

Syntax

PrintY (*printjobnumber*)

Parameter	Description
printjobnumber	The number the PrintOpen function assigned to the print job

Return value

Integer. Returns the y coordinate of the cursor if it succeeds and *-1* if an error occurs.

Example

These statements print a bitmap one inch below the location of the print cursor:

```
integer LocX, LocY, Job

Job = PrintOpen( )
. . .  //Print statements
LocX = PrintX(Job)
LocY = PrintY(Job) + 1000
PrintBitmap(Job, "CORP.BMP", LocX, LocY, 1000,1000)
```

See also PrintX

ProfileInt

Description Obtains the integer value of a setting in the profile file for your application.

Syntax **ProfileInt** (*filename, section, key, default*)

Parameter	Description
filename	A string whose value is the name of the profile file. If you do not specify a full path, ProfileInt uses the operating system's standard file search order to find the file.
section	A string whose value is the name of a group of related values in the profile file. In the file, section names are in square brackcts. Do not include the brackets in *section*. *Section* is not case-sensitive.
key	A string specifying the setting name in *section* whose value you want. The setting name is followed by an equal sign in the file. Do not include the equal sign in *key*. *Key* is not case-sensitive.
default	An integer value that ProfileInt will return if *filename* is not found, if *section* or *key* does not exist in *filename*, or if the value of *key* cannot be converted to an integer.

Return value Integer. Returns *default* if *filename* is not found, *section* is not found in *filename*, or *key* is not found in *section*, or the value of *key* is not an integer. Returns *-1* if an error occurs.

Usage Use ProfileInt or ProfileString to get configuration settings from a profile file that you've designed for your application.

You can use SetProfileString to change values in the profile file to customize your application's configuration during execution. Before you make changes, you can use ProfileInt and ProfileString to obtain the original settings so you can restore them when the user exits the application.

Examples

These examples use a hypothetical file called PROFILE.INI, which contains the following:

```
[Pb]
Maximized=1
[security]
Class=7
```

This statement returns the integer value for the keyword Maximized in section PB of file PROFILE.INI. If there were no PB section or no Maximized keyword in the PB section, it would return 3:

```
ProfileInt("C:\PROFILE.INI", "PB", "maximized", 3)
```

The following statements display a MessageBox if the integer value for the Class setting in section Security of file C:\PROFILE.INI is less than 10. The default security setting is 6 if the profile file is not found or doesn't contain a Class setting:

```
IF ProfileInt("C:\PROFILE.INI", "Security",  &
      "Class", 6) < 10 THEN
   // Class is < 10
   MessageBox("Warning", "Access Denied")
ELSE
   . . .  // Some processing
END IF
```

See also

ProfileString
SetProfileString
ProfileInt in Chapter 2, "DataWindow Painter Functions"

ProfileString

Description

Obtains the string value of a setting in the profile file for your application.

Syntax

ProfileString (*filename*, *section*, *key*, *default*)

Parameter	Description
filename	A string whose value is the name of the profile file. If you do not specify a full path, ProfileString uses the operating system's standard file search order to find the file.
section	A string whose value is the name of a group of related values in the profile file. In the file, section names are in square brackets. Do not include the brackets in *section*. *Section* is not case-sensitive.
key	A string specifying the setting name in *section* whose value you want. The setting name is followed by an equal sign in the file. Do not include the equal sign in *key*. *Key* is not case-sensitive.
default	A string value that ProfileString will return if *filename* is not found, if *section* or *key* does not exist in *filename*, or if the value of *key* cannot be converted to an integer.

Return value

String, with a maximum length of 4096 characters. Returns the string from *key* within *section* within *filename*. If *filename* is not found, *section* is not found in *filename*, or *key* is not found in *section*, ProfileString returns *default*. If an error occurs, it returns the empty string ("").

Usage

Use ProfileInt or ProfileString to get configuration settings from a profile file that you've designed for your application.

You can use SetProfileString to change values in the profile file to customize your application's configuration during execution. Before you make changes, you can use ProfileInt and ProfileString to obtain the original settings so you can restore them when the user exits the application.

Initialization files on the Macintosh
Initialization (INI) files are used extensively to provide setup information for Windows applications. PowerBuilder also uses INI files on the Macintosh. You can use the same INI files on the Macintosh. However, because the Macintosh and Windows handle line breaks differently in text files, you must convert the text file to Macintosh format before you can use it. (The Macintosh uses a single linefeed to end a line, instead of a linefeed and a carriage return).

Examples

These examples use a hypothetical file called PROFILE.INI, which contains the following:

```
[Employee]
Name="Smith"
[Dept]
Name="Marketing"
```

This statement returns the string contained in keyword Name in section Employee in file C:\PROFILE.INI and returns None if there is an error. For the PROFILE.INI file above, the return value is Smith:

```
ProfileString("C:\PROFILE.INI", "Employee",  &
   "Name", "None")
```

On the Macintosh, the following command finds the same information in the PROFILE.INI file the PowerBuilder folder on the drive Hard Disk:

```
ProfileString("Hard Disk:PowerBuilder:PROFILE.INI",&
   "Employee", "Name", "None")
```

The following statements open w_marketing if the string in the keyword Name in section Department of file C:\PROFILE.INI is Marketing:

```
IF ProfileString("C:\PROFILE.INI", "Department",  &
   "Name", "None") = "Marketing" THEN
   Open(w_marketing)
END IF
```

See also

ProfileInt
SetProfileString
ProfileString in Chapter 2, "DataWindow Painter Functions"

Rand

Description

Obtains a random whole number between 1 and a specified upper limit.

Syntax

Rand (*n*)

Parameter	Description
n	The upper limit of the range of random numbers you want returned. The lower limit is always 1. The upper limit is 32,767.

Return value

A numeric data type, the data type of *n*. Returns a random whole number between 1 and *n*, inclusive.

Usage

The sequence of numbers generated by repeated calls to the Rand function is a pseudo-random sequence. You can control whether the sequence is different each time your application runs by calling the Randomize function to initialize the random number generator.

Example

This statement returns a random whole number between 1 and 10:

```
Rand(10)
```

See also

Randomize
Rand in Chapter 2, "DataWindow Painter Functions"

Randomize

Description

Initializes the random number generator so that the Rand function begins a new series of pseudo-random numbers.

Syntax

Randomize (*n*)

Parameter	Description
n	The starting value (seed) for the random number generator. When *n* is 0, PowerBuilder takes the seed from the system clock and begins a nonrepeatable sequence. A nonzero number generates a different but repeatable sequence for each seed value. *n* cannot exceed 32,767.

Return value

Integer. The return value is never used.

Usage

The sequence of numbers generated by repeated calls to the Rand function is a computer-generated pseudo-random sequence. You can use the Randomize function to initialize the random number generator with a value from the system clock, or some other changing value, so that the sequence is always different. For testing purposes, you can select a specific seed value, which you can reuse to make the pseudo-random sequence repeatable each time you run the application.

Include Randomize in the script for the Open event in the application.

Examples

This statement sets the seed for the random number generator to 0 so that calls to Rand generate a new sequence each time the script is run:

```
Randomize(0)
```

This statement sets the seed for the random number generator to 4 so that calls to Rand repeat a specific sequence each time the random number generator is initialized:

```
Randomize(4)
```

See also

Rand

Read

Description

Reads data from an opened OLE stream object into a string (Syntax 1), or a character array or blob (Syntax 2).

Applies to

OLEStream objects

Syntax 1

olestream.**Read** (*variable* {, *stopforline* })

Parameter	Description
olestream	The name of an OLE stream variable that has been opened.
variable	The name of a string variable into which want to read data from *olestream*.

stopforline (optional)	A boolean value that specifies whether to read a line at a time. In other words, Read will stop reading at the next carriage return/linefeed. Values are; ♦ TRUE — (Default) Stop at the end of a line and leave the read pointer positioned after the carriage return/linefeed so the next read will read the next line ♦ FALSE — Read the whole stream or a maximum of 32765 characters

Return value 1

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Stream is not open
- *-2* Read error
- *-9* Other error

Syntax 2

olestream.**Read** (*variable* {, *maximumread* })

Parameter	Description
olestream	The name of an OLE stream variable that has been opened.
variable	The name of a blob variable or character array into which want to read data from *olestream*.
maximumread (optional)	A long value specifying the maximum number of bytes to be read. The default is 32,765 or the length of *olestream*.

Return value 2

Integer. Same as Syntax 1.

Example

This example opens an OLE object in the file MYSTUFF.OLE and assigns it to the OLEStorage object stg_stuff. Then it opens the stream called info in stg_stuff and assigns it to the stream object olestr_info. Finally, it reads the contents of olestr_info into the blob lb_info.

The example doesn't check the functions' return values for success, but you should be sure to check the return values in your code:

```
boolean lb_memexists
OLEStorage stg_stuff
OLEStream olestr_info
blob lb_info

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info",  &
      stgRead!, stgExclusive!)
olestr_info.Read(lb_info)
```

See also

Open
Length
Seek
Write

Real

Description

Converts a string value to a real data type or obtains a real value that is stored in a blob.

Syntax

Real (*stringorblob*)

Parameter	Description
stringorblob	The string whose value you want returned as a real value or a blob in which the first value is the real value. The rest of the contents of the blob is ignored.

Return value

Real. Returns the value of *stringorblob* as a real. If *stringorblob* is not a valid PowerScript number, Real returns *0*.

Examples

This statement returns 24 as a real:

```
Real("24")
```

This statement returns the contents of the SingleLineEdit sle_Temp as a real:

```
Real(sle_Temp.Text)
```

The following example, allow of no practical value, illustrates how to assign real values to a blob and how to use Real to extract those values. The two BlobEdit statements store two real values in the blob, one after the other. In the statements that use Real to extract the values, you have to know where the beginning of each real value is. Specifying the correct length in BlobMid is not important because the Real function knows how many bytes to evaluate:

```
blob{20} lb_blob
real r1, r2
integer len1, len2

len1 = BlobEdit(lb_blob, 1, 32750E0)
len2 = BlobEdit(lb_blob, len1, 43750E0)

// Extract the real value at the beginning and
// ignore the rest of the blob
r1 = Real(lb_blob)
// Extract the second real value stored in the blob
r2 = Real(BlobMid(lb_blob, len1, len2 - len1))
```

See also

Double
Integer
Long
Real in Chapter 2, "DataWindow Painter Functions"

RelativeDate

Description

Obtains the date that occurs a specified number of days after or before another date.

Syntax

RelativeDate (*date*, *n*)

Parameter	Description
date	A date value
n	An integer indicating a number of days

Return value

Date. Returns the date that occurs *n* days after *date* if *n* is greater than 0. Returns the date that occurs *n* days before *date* if *n* is less than 0.

Examples

This statement returns 1990-02-10:

```
RelativeDate(1990-01-31, 10)
```

This statement returns 1990-01-21:

```
RelativeDate(1990-01-31, -10)
```

See also

DaysAfter
RelativeDate in Chapter 2, "DataWindow Painter Functions"

RelativeTime

Description

Obtains a time that occurs a specified number of seconds after or before another time within a 24-hour period.

Syntax

RelativeTime (*time*, *n*)

Parameter	Description
time	A time value
n	A long number of seconds

Return value

Time. Returns the time that occurs *n* seconds after *time* if *n* is greater than 0. Returns the time that occurs *n* seconds before *time* if *n* is less than 0. The maximum return value is 23:59:59.

Examples

This statement returns 19:01:41:

```
RelativeTime(19:01:31, 10)
```

This statement returns 19:01:21:

```
RelativeTime(19:01:31, -10)
```

See also

SecondsAfter
RelativeTime in Chapter 2, "DataWindow Painter Functions"

Repair

Description

Updates the target database with corrections that have been made in the pipeline user object's Error DataWindow.

Applies to

Pipeline objects

pipelineobject.**Repair** (*destinationtrans*)

Parameter	Description
pipelineobject	The name of a pipeline user object that contains the pipeline object being executed
destinationtrans	The name of a transaction object with which to connect to the target database

Return value

Integer. Returns *1* if it succeeds and a negative number if an error occurs. Error values are:

- *-5* Missing connection
- *-9* Fatal SQL error in destination
- *-10* Maximum number of errors exceeded
- *-11* Invalid window handle
- *-12* Bad table syntax
- *-15* Pipe already in progress
- *-17* Error in destination database
- *-18* Destination database is read-only

Usage

When errors have occurred during a pipeline data transfer, Start populates its pipeline-error DataWindow control with the rows that caused the errors. The user or a script can then make corrections to the data. The Repair function is usually associated with a CommandButton that the user can click after correcting data in the pipeline-error DataWindow.

If errors occur again, the rows that are in error remain in the pipeline-error DataWindow. The user can correct the data again and click the button that calls Repair.

Example

This statement connects to the destination database using the transaction instance variable i_dst. It then updates the database with the corrections made in the Error DataWindow for pipeline i_pipe:

```
i_pipe.Repair(i_dst)
```

See also

Cancel
Repair
Start

Replace

Description

Replaces a portion of one string with another.

Syntax

Replace (*string1, start, n, string2*)

Parameter	Description
string1	The string in which you want to replace characters with *string2*.
start	A long whose value is the number of the first character you want replaced. (The first character in the string is number 1.)
n	A long whose value is the number of characters you want to replace.
string2	The string that will replace characters in *string1*. The number of characters in *string2* can be greater than, equal to, or less than the number of characters you are replacing.

Return value

String. Returns the string with the characters replaced if it succeeds and the empty string if it fails.

Usage

If the start position is beyond the end of the string, Replace appends *string2* to *string1*. If there are fewer characters after the start position than specified in *n,* Replace replaces all the characters to the right of character *start*.

If *n* is zero, then, in effect, Replace inserts *string2* into *string1*.

Examples

These statements change the value of Name from Davis to Dave:

```
string Name
Name = "Davis"
Name = Replace(Name, 4, 2, "e")
```

This statement returns BABY RUTH:

```
Replace("BABE RUTH", 1, 4, "BABY")
```

This statement returns Closed for the Winter:

```
Replace("Closed for Vacation", 12, 8, "the Winter")
```

This statement returns ABZZZZEF:

```
Replace("ABCDEF", 3, 2, "ZZZZ")
```

This statement returns ABZZZZ:

```
Replace("ABCDEF", 3, 50, "ZZZZ")
```

This statement returns ABCDEFZZZZ:

```
Replace("ABCDEF", 50, 3, "ZZZZ")
```

These statements replace all occurrences of red within the string mystring with green. The original string is taken from the SingleLineEdit sle_1 and the result becomes the new text of sle_1:

```
long start_pos=1
string old_str, new_str, mystring

mystring = sle_1.Text
old_str = "red"
new_str = "green"

// Find the first occurrence of old_str.
start_pos = Pos(mystring, old_str, start_pos)

// Only enter the loop if you find old_str.
DO WHILE start_pos > 0

   // Replace old_str with new_str.
   mystring = Replace(mystring, start_pos,  &
      Len(old_str), new_str)

   // Find the next occurrence of old_str.
   start_pos = Pos(mystring, old_str,  &
      start_pos+Len(new_str))
LOOP

sle_1.Text = mystring
```

See also

Replace in Chapter 2, "DataWindow Painter Functions"

ReplaceText

Description

Replaces selected text in an edit control with a specified string.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, and DropDownListBox controls

Syntax

editname.**ReplaceText** (*string*)

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit or SingleLineEdit, or DropDownListBox in which you want to replace the selected string. In a DataWindow control, the text is replaced in the edit control over the current row and column.
string	The string that replaces the selected text.

Return value

Integer. Returns the number of characters in *string* and *-1* if an error occurs.

Usage

If there is no selected text, ReplaceText inserts the replacement text at the cursor position.

Tips

To use the contents of the clipboard as the replacement text, call the Paste function, instead of ReplaceText.

To replace text in a string, rather than a control, use the Replace function.

Examples

If the MultiLineEdit mle_Comment contains Offer Good for 3 Months and the selected text is 3 Months, this statement replaces 3 Months with 60 Days and returns 7. The resulting value of mle_Comment is Offer Good for 60 Days.

```
mle_Comment.ReplaceText("60 Days")
```

If there is no selected text, this statement inserts "Draft" at the cursor position in the SingleLineEdit sle_Comment3:

```
sle_Comment3.ReplaceText("Draft")
```

See also

Copy

Cut
Paste

ReselectRow

Description

Accesses the database to retrieve values for all columns that can be updated and refreshes all timestamp columns in a row in a DataWindow control. The values from the database are redisplayed in the row.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ReselectRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to reselect a row
row	A long identifying the row to reselect

Return value

Integer. Returns *1* if it is successful and *-1* if the row cannot be reselected (for example, the DataWindow object cannot be updated or the row was deleted by another user).

Usage

Use ReselectRow to discard values the user changed and replace them with values from the database after an update fails due to a timestamp error.

> **Note**
> Timestamp support is not available in all DBMSs. For information on timestamp columns, see the documentation for your DBMS.

Examples

This statement reselects row 5 in the DataWindow control dw_emp:

```
dw_emp.ReselectRow(5)
```

This statement reselects the clicked row if the update is not successful:

```
IF dw_emp.Update( ) < 0 THEN
   dw_emp.ReselectRow(dw_emp.GetClickedRow( ))
END IF
```

See also

GetClickedRow
SelectRow
Update

Reset

Description

Clears data from a control. There are three syntaxes, depending on the target object:

- To clear the data from a DataWindow control, use Syntax 1.
- To delete all items from a list, use Syntax 2.
- To delete all the data, and, optionally, the series and categories, from a graph, use Syntax 3.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) ListBox and DropDownListBox controls

(Syntax 3) Graph controls in windows and user objects. Does not apply to graphs within DataWindow objects because their data comes directly from the DataWindow.

Syntax 1

datawindowname.**Reset** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you want to clear

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Syntax 2

listboxname.**Reset** ( )

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox from which to delete all items

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Syntax 3

controlname.**Reset** (*graphresettype*)

Parameter	Description
controlname	The name of the graph object in which you want to delete all the data values or all series and all data values
graphresettype	A value of the grResetType enumerated data type specifying whether you want to delete only data values or all series and all data values: ♦ All! — Delete all series, categories, and data in *controlname* ♦ Category! — Delete categories and data in *controlname* ♦ Data! — Delete data in *controlname* ♦ Series! — Delete the series and data in *controlname*

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

Reset (Syntax 1) is not the same as deleting rows from the DataWindow object or child DataWindow. Reset affects the application only, not the database. If you delete rows and then call the Update function, the rows are deleted from the database table associated with the DataWindow object. If you call Reset and then call Update, no changes are made to the table.

Use Syntax 3 of Reset to clear the data in a graph before you add new data.

Examples

This statement completely clears the contents of dw_employee:

```
dw_employee.Reset( )
```

Syntax 2

This statement deletes all items in the ListBox portion of ddlb_Actions:

```
ddlb_Actions.Reset( )
```

Syntax 3

This statement deletes the series and data, but leaves the categories, in the graph gr_product_data:

```
gr_product_data.Reset(Series!)
```

See also

AddData
AddSeries

DeleteRow
DeleteItem

ResetDataColors

Description

Restores the color of a data point to the default color for its series.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**ResetDataColors** ({*graphcontrol*, } *seriesnumber*, & *datapointnumber*)

Parameter	Description
controlname	The name of the graph in which you want to reset the color of a data point, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to reset the color
seriesnumber	The number of the series in which you want to reset the color of a data point
datapointnumber	The number of the data point for which you want to reset the color

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

> **Tip**
> To set the color for a series, use SetSeriesStyle. The color you set for the series is the default color for all data points in the series.

Examples

These statements change the color of data point 10 in the series named Costs in the graph gr_product_data to the color for the series:

```
SeriesNbr = gr_product_data.FindSeries("Costs")
gr_product_data.ResetDataColors(SeriesNbr, 10)
```

These statements change the color of data point 10 in the series named Costs in the graph gr_computers in the DataWindow control dw_equipment to the color for the series:

```
SeriesNbr = dw_equipment.FindSeries("Costs")
dw_equipment.ResetDataColors("gr_computers",  &
   SeriesNbr, 10)
```

See also

GetDataStyle
SeriesName
GetSeriesStyle
SetDataStyle
SetSeriesStyle

ResetTransObject

Description

Stops a DataWindow control from using the programmer-specified transaction object that is currently in effect via a call the SetTransObject. After you call the ResetTransObject function, the DataWindow control uses its internal transaction object.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ResetTransObject** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to stop using a programmer-specified transaction object and begin using the DataWindow control's internal transaction object

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

If you reset the transaction object and SetTrans has never been called to set the values in the DataWindow control's internal transaction object, call SetTrans to set them or SetTransObject to establish a new programmer-specified transaction object.

ResetTransObject is almost never used because you generally do not mix the use of programmer-specified and internal transaction objects in one application. Programmer-specified transaction objects, specified with SetTransObject, provide better application performance. To change the programmer-specified transaction object, simply call SetTransObject again.

Example

This statement stops dw_employee from using programmer-specified transaction objects:

```
dw_employee.ResetTransObject( )
```

See also

GetTrans
SetTrans
SetTransObject

ResetUpdate

Description

Clears the update flags in the primary and filter buffers and empties the delete buffer of the DataWindow.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ResetUpdate** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to reset the update flags

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

When a row is changed, inserted, or deleted, its update flag is set, making it marked for update. By default the Update function turns these flags off. However, if you want to coordinate updates of more than one DataWindow, you can prevent Update from clearing the flags. Then after you verify that all the updates succeeded, you can call ResetUpdate for each DataWindow to clear the flags. If one of the updates failed, you can keep the update flags, prompt the user to fix the problem, and try the updates again.

You can find out which rows are marked for update with the GetItemStatus function. If a row is in the delete buffer or if it is in the primary or filter buffer and has NewModified! or DataModified! status, its update flag is set. After update flags are cleared, all rows have the status NotModified! or New! and the delete buffer is empty.

Example

These statements coordinate the update of two DataWindow objects:

```
int rtncode

CONNECT USING SQLCA;
dw_cust.SetTransObject(SQLCA)
dw_sales.SetTransObject(SQLCA)

rtncode = dw_cust.Update(TRUE, FALSE)
IF rtncode = 1  THEN
   rtncode = dw_sales.Update(TRUE, FALSE)
   IF rtncode = 1  THEN
      dw_cust.ResetUpdate( )  // Both updates are OK
      dw_sales.ResetUpdate( ) // Clear update flags
      COMMIT USING SQLCA;     // Commit them
   ELSE
      ROLLBACK USING SQLCA;   // 2nd update failed
   END IF
END IF
```

See also

Update

Resize

Description

Resizes an object or control by setting its width and height attributes and then redraws the object.

Applies to

Any object, except a child DataWindow

Syntax

objectname.**Resize** (*width*, *height*)

Parameter	Description
objectname	The name of the object or control you want to resize
width	The new width in PowerBuilder units
height	The new height in PowerBuilder units

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

You cannot use Resize for a child DataWindow.

Equivalent syntax

You can set the object's Width and Height attributes instead of calling the Resize function, as shown below. However, the two statements cause PowerBuilder to redraw *objectname* twice; first with the new width, and then with the new width and height.

objectname.Width = *width*
objectname.Height = *height*

These statements, although they redraw gb_box1 twice:

```
gb_box1.Width = 100
gb_box1.Height = 150
```

achieve the same result as:

```
gb_box1.Resize(100, 150)
```

Examples

This statement changes the Width and Height attributes of gb_box1 and redraws gb_box1 with the new attributes:

```
gb_box1.Resize(100, 150)
```

This statement doubles the width and height of the picture control p_1:

```
p_1.Resize(p_1.Width*2, p_1.Height*2)
```

RespondRemote

Description

Sends a DDE message indicating whether the command or data received from a remote DDE application was acceptable.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

RespondRemote (*boolean*)

Parameter	Description
boolean	A boolean expression. TRUE indicates that the previously received command or data was acceptable. FALSE indicates that it was not.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs (for example, the function was called in wrong context).

Usage

You can use RespondRemote when the PowerBuilder application is the DDE server or DDE client application.

You usually call RespondRemote after these functions:

- GetCommandDDE
- GetCommandDDEOrigin
- GetDataDDE
- GetDataDDEOrigin

For more information about PowerBuilder as a client, see OpenChannel and ExecRemote. For more information about PowerBuilder as a server, see StartServerDDE.

Example

In a script for the HotLinkAlarm event, these statements tell a remote application named Gateway that its data was successfully received:

```
String Applname, Topic, Item, Value

GetDataDDEOrigin(Applname, Topic, Item)

IF Applname = "Gateway" THEN
   IF GetDataDDE(Value) = 1 THEN
      RespondRemote(TRUE)
   END IF
END IF
```

See also

GetCommandDDE
GetCommandDDEOrigin
GetDataDDE
GetDataDDEOrigin

Restart

Description

Stops the execution of all scripts, closes all windows (without executing the scripts for the Close events), commits and disconnects from the database, restarts the application, and executes the application-level script for the Open event.

Syntax

Restart ( )

Return value

Integer. Returns *1* if it succeeds and *-1* if it fails. The return value is usually not used.

Usage

You can use Restart in the application-level script for the Idle event to restart the application after a period of user inactivity, a typical behavior of kiosk applications.

Example

In the application-level script for the Idle event, this statement restarts the application:

```
Restart( )
```

See also

HALT in *PowerScript Language*

Retrieve

Description

Retrieves rows from the database for a DataWindow control. If arguments are included, the argument values are used for the retrieval arguments in the SQL SELECT statement for the DataWindow object or child DataWindow.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Retrieve** ({, *argument*, *argument* . . . })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that you want to cause to retrieve rows from the database
argument (optional)	One or more values that you want to used as retrieval arguments in the SQL SELECT statement defined in *datawindowname*

Return value

Long. Returns the number of rows displayed (that is, rows in the primary buffer) if it succeeds and *-1* if it fails.

Usage

After rows are retrieved, the DataWindow object's filter is applied. Therefore, any retrieved rows that don't meet the filter criteria are immediately moved to the filter buffer and are not included in the return count.

Before you can retrieve rows for a DataWindow control, you must specify a transaction object with SetTransObject or SetTrans. If you use SetTransObject, you must also use an SQL CONNECT statement to establish a database connection.

Normally, when you call Retrieve, any rows that are already in the DataWindow control are discarded and replaced with the retrieved rows. You can set the action code for the RetrieveStart event to 2 to prevent this. In this case, Retrieve adds any retrieved rows to the ones that are already in the DataWindow's buffers. (See the last example.)

Events

Retrieve may trigger these events:

- DBError
- RetrieveEnd
- RetrieveRow
- RetrieveStart

Examples

This statement causes dw_emp1 to retrieve rows from the database:

```
dw_emp1.Retrieve( )
```

This example illustrates how to set up a connection and then retrieve rows in the DataWindow control. A typical scenario is to establish the connection in the application's Open event and to retrieve rows in the Open event for the window that contains the DataWindow control.

The following is a script for the application open event. SQLCA is the default transaction object. The ProfileString function is getting information about the database connection from an initialization file:

```
// Set up Transaction object from the INI file
SQLCA.DBMS = ProfileString("PB.INI",  &
   "Database", "DBMS", "  ")
SQLCA.DbParm = ProfileString("PB.INI",  &
   "Database", "DbParm", "  ")

// Connect to database
CONNECT USING SQLCA;

// Test whether the connect succeeded
IF SQLCA.SQLCode <> 0 THEN
  MessageBox("Connect Failed",  &
    "Cannot connect to database. " +
SQLCA.SQLErrText)
  RETURN
END IF
Open(w_main)
```

To continue the example, the open event for w_main sets the transaction object for the DataWindow control dw_main to SQLCA and retrieves rows from the database. If no rows were retrieved or if there is an error (that is, the return value is negative), the script displays a message to the user:

```
long ll_rows
dw_main.SetTransObject( SQLCA )

ll_rows = dw_main.Retrieve( )
IF ll_rows < 1 THEN MessageBox(  &
   "Database Error",  &
   "No rows retrieved." )
```

This example illustrates the use of retrieval arguments. Assume dw_emp contains this SQL SELECT statement:

```
SELECT Name, emp.sal, sales.rgn From Employee
   WHERE emp.sal > :Salary and sales.rgn = :Region
```

Note

:Salary and :Region are declared as arguments in the DataWindow painter. To modify the SQL SELECT statement, choose Edit Data Source from the Design menu in the DataWindow painter.

Then this statement causes dw_emp1 to retrieve employees from the database who have a salary greater than $50,000 and are in the northwest region:

```
dw_emp1.Retrieve(50000, "NW")
```

This example also illustrates retrieval arguments. Assume dw_EmpHist contains this SQL SELECT statement and emps is defined as a number array:

```
SELECT EmpNbr, Sal, Rgn From Employee
   WHERE EmpNbr IN (:emps)
```

These statements cause dw_EmpHist to retrieve Employees from the database whose employee numbers are values in the array emps:

```
Double emps[3]
emps[1] = 100
emps[2] = 200
emps[3] = 300
dw_EmpHist.Retrieve(emps)
```

This example illustrates how to use Retrieve twice to get data meeting different criteria. Assume the SELECT statement for the DataWindow object requires one argument, the department number. Then these statements retrieve all rows in the database in which department number is 100 or 200.

The script for the RetrieveStart event in the DataWindow control sets the return code to 2 so the rows and buffers of the DataWindow control will not be cleared before each retrieval:

```
This.SetActionCode(2)
```

The script for the Clicked event for a Retrieve CommandButton retrieves the data with two function calls. The Reset function clears any previously retrieved rows, normally done by Retrieve. Here, Retrieve is prevented from doing it by the action code setting in the RetrieveStart event:

```
dw_1.Reset( )
dw_1.Retrieve(100)
dw_1.Retrieve(200)
```

See also

DeleteRow
InsertRow
SetTrans
SetTransObject

RGB

Description

Calculates the long value that represents the color specified by numeric values for the red, green, and blue components of the color.

Syntax

RGB (*red, green, blue*)

Parameter	Description
red	The integer value of the red component of the desired color
green	The integer value of the green component of the desired color
blue	The integer value of the blue component of the desired color

Return value

Long. Returns the long that represents the color created by combining the values specified in red, green, and blue. If an error occurs, RGB returns *-1*.

Usage

The formula for combining the colors is:

```
65536 * Blue + 256 * Green + Red
```

Use RGB to obtain the long value required to set the color for text and drawing objects. You can also set an object's color to the long value that represents the color. The RGB function provides an easy way to calculate that value.

> **Tip**
> The value of a component color is an integer between 0 and 255 that represents the amount of the color that is required to create the color you want. The lower the value, the darker the color; the higher the value, the lighter the color.
>
> To determine the values for the components of a color (known as the RGB values), use the Edit Color Entry window. To access the Edit Color Entry window, select a color in the color bar at the bottom of the workspace and then double-click the selected color when it displays in the first box of the color bar.

The following table lists red, green, and blue values for the 16 "standard" colors:

Color	Red Value	Green Value	Blue Value
Black	0	0	0
White	255	255	255
Light Gray	192	192	192
Dark Gray	128	128	128
Red	255	0	0
Dark Red	128	0	0
Green	0	255	0
Dark Green	0	128	0
Blue	0	0	255
Dark Blue	0	0	128
Magenta	255	0	255
Dark Magenta	128	0	128
Cyan	0	255	255
Dark Cyan	0	128	128
Yellow	255	255	0
Brown	128	128	0

Examples

This statement returns a long that represents black:

```
RGB(0, 0, 0)
```

This statement returns a long that represents white:

```
RGB(255, 255, 255)
```

These statements set the color attributes of the StaticText st_title to be green letters on a dark magenta background:

```
st_title.TextColor = RGB(0, 255, 0)
st_title.BackColor = RGB(128, 0, 128)
```

Right

Description

Obtains a specified number of characters from the end of a string.

Syntax

Right (*string, n*)

Parameter	Description
string	The string from which you want characters returned
n	A long whose value is the number of characters you want returned from the right end of *string*

Return value

String. Returns the rightmost *n* characters in *string* if it succeeds and the empty string ("") if an error occurs.

If *n* is greater than or equal to the length of the string, Right returns the entire string. It does not add spaces to make the return value's length equal to *n*.

Examples

This statement returns RUTH:

```
Right("BABE RUTH", 4)
```

This statement returns BABE RUTH:

```
Right("BABE RUTH", 75)
```

See also

Left
Mid
Pos
Right in Chapter 2, "DataWindow Painter Functions"

RightTrim

Description

Removes spaces from the end of a string.

Syntax

RightTrim (*string*)

Parameter	Description
string	The string you want returned with trailing blanks deleted

Return value

String. Returns a copy of *string* with trailing blanks deleted if it succeeds and the empty string ("") if an error occurs.

Example

This statement returns RUTH:

```
RightTrim("RUTH    ")
```

See also

LeftTrim
Trim
RightTrim in Chapter 2, "DataWindow Painter Functions"

Round

Description

Rounds a number to the specified number of decimal places.

Syntax

Round (*x*, *n*)

Parameter	Description
x	The number you want to round.
n	The number of decimal places to which you want to round *x*. Valid values are 0 through 18.

Return value

Decimal. Returns *x* rounded to the specified number of decimal places if it succeeds, and NULL if it fails.

Examples

This statement returns 9.62:

```
Round(9.624, 2)
```

This statement returns 9.63:

```
Round(9.625, 2)
```

This statement returns 9.600:

```
Round(9.6, 3)
```

This statement returns –9.63:

```
Round(-9.625, 2)
```

This statement returns NULL:

```
Round(-9.625, -1)
```

See also

Ceiling
Int
Truncate
Round in Chapter 2, "DataWindow Painter Functions"

RowCount

Description

Obtains the number of rows that are currently available for display in a DataWindow control, in other words, the number of rows in the DataWindow's primary buffer.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**RowCount ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want the number of rows currently available for display.

Return value

Long. Returns the number of rows that are currently available in *datawindowname*, *0* if no rows are currently available, and *-1* if an error occurs.

Usage

The DataWindow control's primary buffer contains the rows that are currently available for display. These are the rows counted by RowCount. The number of currently available rows equals the total number of rows retrieved minus any deleted rows plus any inserted rows minus any rows that have been filtered out. The deleted and filtered rows are stored in the DataWindow's delete and filter buffers.

Example

This statement returns the number of rows currently available in dw_Employee:

```
long NbrRows
NbrRows = dw_Employee.RowCount( )
```

This example determines when the user has scrolled to the end of a DataWindow control. It compares the row count with the DataWindow attribute LastRowOnPage:

```
dw_1.ScrollNextPage()
IF dw_1.RowCount( ) = Integer(dw_1.Describe(  &
   "DataWindow.LastRowOnPage")) THEN
   . . . // Appropriate processing
END IF
```

See also

DeleteRow
DeletedCount
Filter
FilteredCount
InsertRow
ModifiedCount
SetFilter
Update

RowsCopy

Description

Copies a range of rows from one DataWindow control to another or from one buffer to another within a single DataWindow.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**RowsCopy** (*startrow*, *endrow*, *copybuffer*, & *targetdw*, *beforerow*, *targetbuffer*)

Parameter	Description
datawindowname	The name of a DataWindow control or child DataWindow from which you want to copy rows.
startrow	A long whose value is the number of the first row you want to copy.
endrow	A long whose value is the number of the last row you want to copy.

Parameter	Description
copybuffer	A value of the dwBuffer enumerated data type specifying the buffer from which you want to copy the rows: ♦ Primary! ♦ Delete! ♦ Filter!
targetdw	The name of the DataWindow control to which you want to copy the rows. *Targetdw* can be the same DataWindow control or another DataWindow control.
beforerow	A long specifying the number of the row before which you want to insert the copied rows. To insert after the last row, use any value that is greater than the number of existing rows.
targetbuffer	A value of the dwBuffer enumerated data type specifying the target buffer for the copied rows. Values are: ♦ Primary! ♦ Delete! ♦ Filter!

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The status of rows that are copied to a DataWindow's primary buffer is NewModified!. If you update the DataWindow control, PowerBuilder will send SQL INSERT statements to the DBMS for the rows.

Uses for RowsCopy include:

- Making copies of one or more rows so that the users can make create new rows based on existing data
- Printing a range of rows by copying selected rows to another DataWindow and printing the second DataWindow control

Example

This statement copies all the rows starting with the current row in dw_1 to the beginning of the primary buffer in dw_2:

```
dw_1.RowsCopy(dw_1.GetRow(),  &
   dw_1.RowCount(), Primary!, dw_2, 1, Primary!)
```

See also

RowsDiscard
RowsMove

RowsDiscard

Description

Discards a range of rows in a DataWindow control. Once a row has been discarded using RowsDiscard, you cannot restore the row—you have to retrieve it again from the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**RowsDiscard** (*startrow*, *endrow*, *buffer*)

Parameter	Description
datawindowname	The name of a DataWindow control or child DataWindow from which you want to discard rows.
startrow	A long whose value is the number of the first row you want to discard.
endrow	A long whose value is the number of the last row you want to discard.
buffer	A value of the dwBuffer enumerated data type specifying the buffer containing the rows to be discarded. Values are: ♦ Primary! ♦ Delete! ♦ Filter!

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use RowsDiscard when your application is finished with some of the rows in a DataWindow control and you don't want an update to affect the rows in the database. For example, you can discard rows in the delete buffer, which prevents the rows from being deleted when you call Update.

To clear all the rows from a DataWindow control, use Reset.

Example

This statement discards all the rows in the delete buffer for dw_1. As a result if the application later calls dw_1.Update(), the DataWindow will not submit SQL DELETE statements to the DBMS for these rows:

```
dw_1.RowsDiscard(1, dw_1.DeletedCount(), Delete!)
```

See also

Reset
RowsCopy
RowsMove

RowsMove

Description

Clears a range of rows from a DataWindow control and inserts the rows in another DataWindow control or another buffer of the same DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**RowsMove** (*startrow*, *endrow*, *movebuffer*, & *targetdw*, *beforerow*, *targetbuffer*)

Parameter	Description
datawindowname	The name of a DataWindow control or child DataWindow from which you want to move rows.
startrow	A long whose value is the number of the first row you want to move.
endrow	A long whose value is the number of the last row you want to move.
movebuffer	A value of the dwBuffer enumerated data type specifying the buffer from which you want to move the rows. Values are: ◆ Primary! ◆ Delete! ◆ Filter!
targetdw	The name of the DataWindow control to which you want to move the rows. *Targetdw* can be the same DataWindow control or another DataWindow control.

Parameter	Description
beforerow	A long specifying the number of the row before which you want to insert the moved rows. To insert after the last row, use any value that is greater than the number of existing rows.
targetbuffer	A value of the dwBuffer enumerated data type specifying the target buffer for the moved rows. Values are: ♦ Primary! ♦ Delete! ♦ Filter!

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If you move rows to another DataWindow control, the rows have the status NewModified! in the target DataWindow.

If you move rows between buffers in a single DataWindow control, PowerBuilder retains knowledge of where the rows came from and their status is changed accordingly. For example, if you move unmodified rows from the primary buffer to the delete buffer, they are marked for deletion. If you move the rows back to the primary buffer, their status returns to NotModified!. Note, however, that if you move rows from one DataWindow control to another and back again, the rows' status is NewModified! because they came from a different DataWindow.

Uses for RowsMove include:

- Moving several rows from the primary buffer to the delete buffer, instead of deleting them one at a time
- Moving rows from the delete buffer to the primary buffer, to implement an Undo capability in your application

Example

This statement moves all the rows starting with the first row in the delete buffer for dw_1 to the primary buffer for dw_1; thereby *undeleting* these rows:

```
dw_1.RowsMove(1, dw_1.DeletedCount(), Delete!,  &
   dw_1, 1, Primary!)
```

See also

RowsCopy
RowsDiscard

Run

Description

Runs the specified application program.

Syntax

Run (*string* {, *windowstate* })

Parameter	Description
string	A string whose value is the filename of the program you want to execute. Optionally, *string* can contain one or more parameters for the program.
windowstate (optional)	A value of the WindowState enumerated data type indicating the state in which you want to run the program: ♦ Maximized! — Maximized; enlarge the program window to its maximum size when it starts. ♦ Minimized! — Minimized; shrink the program window to an icon when it starts. Minimized! has no effect on the Macintosh. ♦ Normal! — (Default) Run the program window in its normal size.

Return value

Integer. Returns *1* if it is successful and *-1* if an error occurs.

Usage

You can run any program that you can run from the operating system. If you do not specify parameters, Run opens the application and displays the first application window. If you specify *windowstate*, the application window is displayed in the specified state.

If you specify parameters, the application determines the meaning of those parameters. A typical use is to identify a data file to be opened when the program is executed. If you are running another PowerBuilder application, that application can call the CommandParm function to retrieve the parameters and process them as it sees fit.

If the file extension is omitted from the filename, PowerBuilder assumes the extension is .EXE. To run a program with another extension (for example, .BAT, .COM, or .PIF), you must specify the extension.

Examples

This statement runs the Microsoft Windows 3.x Clock accessory application in its normal size:

```
Run("Clock")
```

This statement runs the Microsoft Windows 3.x Clock accessory application minimized:

```
Run("Clock", Minimized!)
```

In Windows, this statement runs the program WINNER.COM on the C drive maximized and passes it the parameter EMPLOYEE.INF to open the file EMPLOYEE.INF:

```
Run("C:\WINNER.COM EMPLOYEE.INF", Maximized!)
```

This example runs the DOS batch file MYBATCH.BAT and passes the parameter "TEST" to the batch file. In the batch file, you include percent substitution characters in the commands to indicate where the parameter is used:

```
Run("MYBATCH.BAT TEST")
```

In the batch file the following statement renames FILE1 to TEST:

```
RENAME c:\PB4:\FILE1 %1
```

On the Macintosh, this statement runs the Chooser, which is in the Apple Items folder inside the System folder:

```
Run( &
 "Hard Disk:System Folder:Apple Menu Items:Chooser")
```

Tip
If you open a response window from which you run your program, PowerBuilder will wait until the program completes before executing the next line in the script.

Save

Description

Saves an OLE object. There are two syntaxes, one for saving an object in an OLE 2.0 control and another for saving a storage.

Syntax 1

ole2control.**Save** ( )

Parameter	Description
ole2control	The name of the OLE 2.0 control containing the object you want to save

Return value 1

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Control is empty
- *-9* Other error

Syntax 2

ole2storage.**Save** ( )

Parameter	Description
ole2storage	The name of the storage object that you want to save

Return value 2

Integer. Same as Syntax 1.

Usage

When you save an OLE object, PowerBuilder saves it according to the current connection between it and an open storage or file. You establish an initial connection when you call the Open function. When you call SaveAs, the old connection is ended and a new connection is established with another storage or file.

When you call Save for an OLE 2.0 control, PowerBuilder saves the object in the OLE 2.0 control to the storage to which it is currently connected. The storage can be a storage object variable or a OLE storage file.

If the data has never been saved in the server application, so that there is no file on disk, the Save function in PowerBuilder will return an error.

When you call Save for a storage object variable, PowerBuilder saves the storage to the file, or substorage within the file, to the file to which it is currently connected. You must have previously established a connection to an OLE storage file on disk, or a substorage within the file, either with Open or SaveAs.

When do you have to save twice?
If you create a storage object variable and then open that object in an OLE 2.0 control, you will need to call Save twice to write changed OLE information to disk: once to save from the object in the control to the storage, and again to save the storage to its associated file.

Examples

This example saves the object in the control ole_1 back to the storage from which it was loaded, either a storage object variable or a file on disk:

```
integer result
result = ole_1.Save( )
```

This example saves a storage object to its file. Olestor_1 is an instance variable of type olestorage:

```
integer result
result = olestor_1.Save( )
```

In a window's Open script, this code creates a storage variable ole_stor, which is declared as an instance variable, and associates it with a storage file that contains several Visio drawings. The script then opens one of the drawings into the control ole_draw. After the user activates and edits the object, the script for a Save button saves the object to the storage and then to the storage's file.

The script for the window's Open event includes:

```
OLEStorage stg_stor

stg_stor = CREATE OLEStorage
stg_stor.Open("myvisio.ole")
ole_draw.Open(ole_stor, "visio_drawing1")
```

The script for the Save button's Clicked event is:

```
integer result
result = ole_draw.Save( )
IF result = 0 THEN ole_stor.Save( )
```

See also

Close
SaveAs

SaveAs

Description

Saves the contents of a DataWindow, graph, OLE 2.0 control, or OLE storage in a file. For DataWindows and graphs, you can save the data as tab- or comma-delimited text, dBase, Lotus, or Excel files, or as SQL statements, among other formats.

- To save the contents of a DataWindow object, use Syntax 1.
- To save the data in a graph, use Syntax 2.
- To save the OLE object in an OLE 2.0 control to a storage file, use Syntax 3.

- To save the OLE object in an OLE 2.0 control to a storage object in memory, use Syntax 4.
- To save an OLE storage and any controls that have opened that storage in a file, use Syntax 5.
- To save an OLE storage object in another OLE storage object, use Syntax 6.

Applies to

(Syntax 1) DataWindow controls and child DataWindows

(Syntax 2) Graph controls in windows and user objects, and graphs in DataWindow controls

(Syntax 3 and 4) OLE 2.0 controls

(Syntax 5 and 6) OLE storage objects

Syntax 1

datawindowname.**SaveAs** ({ *filename, saveastype, colheading* })

Parameter	Description
datawindowname	The name of the DataWindow control or Child DataWindow whose contents you want to save.
filename (optional)	A string whose value is the name of the file in which to save the contents. If you omit *filename*, PowerBuilder prompts for the filename.
saveastype (optional)	A value of the SaveAsType enumerated data type specifying the format in which to save the contents of the DataWindow object. Values are: ♦ CSV! — Comma-separated values ♦ Clipboard! — Save to the clipboard ♦ dBASE2! — dBASE-II format ♦ dBASE3! — dBASE-III format ♦ DIF! — Data Interchange Format ♦ Excel! — Microsoft Excel format ♦ SQLInsert! — SQL syntax ♦ SYLK! — Microsoft Multiplan format ♦ Text! — (Default) Tab-separated columns with a return at the end of each row ♦ WKS! — Lotus 1-2-3 format ♦ WK1! — Lotus 1-2-3 format

Parameter	Description
colheading (optional)	A boolean value indicating whether you want to include the DataWindow's column headings at the beginning of the file. The default value is TRUE. **Note** Column headings are always saved for dBASE files.

Return value 1 Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2 *controlname*.**SaveAs** ({*graphcontrol*,} { *filename*, *saveastype* , & *colheading* })

Parameter	Description
controlname	The name of the graph control whose contents you want to save or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control whose contents you want to save.
filename (optional)	A string whose value is the name of the file in which you want to save the contents. If you omit *filename*, PowerBuilder prompts the user for a file name.
saveastype	The format in which you want to save the data represented in the graph, specified as a value of the SaveAsType enumerated data type. See the list for *saveastype* in Syntax 1.
colheading	A boolean value indicating whether you want column headings with the saved data. The default value is TRUE. *Colheading* is ignored for dBASE files; column headings are always saved.

Return value 2 Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 3 *ole2control*.**SaveAs** (*OLEtargetfile*)

Parameter	Description
ole2control	The name of the OLE 2.0 control containing the object you want to save.
OLEtargetfile	A string specifying the name of an OLE storage file. The file can already exist. *OLEtargetfile* can include a path, as well as information about where to store the object in the file's internal structure.

Return value 3

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The control is empty
- *-2* The storage is not open
- *-3* The storage name is invalid
- *-9* Other error

Syntax 4

ole2control.**SaveAs** (*targetstorage, substoragename*)

Parameter	Description
ole2control	The name of the OLE 2.0 control containing the object you want to save.
targetstorage	The name of an object variable of OLEStorage in which to store the object in *ole2control.*
substoragename	A string whose value is the name of a substorage within *targetstorage.* If *substorage* does not exist, SaveAs will create it.

Return value 4

Integer. Same as Syntax 3.

Syntax 5

olestorage.**SaveAs** (*OLEtargetfile*)

Parameter	Description
olestorage	The name of an object variable of type OLEStorage containing the OLE object you want to save.
OLEtargetfile	A string specifying the name of a new OLE storage file. *OLEtargetfile* can include a path.

Return value 5

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* The storage is not open
- *-2* The storage name is invalid
- *-3* The parent storage is not open
- *-4* The file already exists

- -5 Insufficient memory
- -6 Too many files open
- -7 Access denied
- -9 Other error

Syntax 6

olestorage.**SaveAs** (*substoragename, targetstorage*)

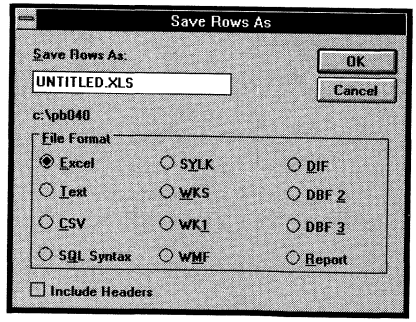
Parameter	Description
olestorage	The name of an object variable of type OLEStorage containing the OLE object you want to save.
substoragename	A string whose value is the name of a substorage within *targetstorage*. If *substorage* does not exist, SaveAs will create it.
targetstorage	The name of an object variable of OLEStorage in which to store the object in *olestorage*. Note the reversed order of the *substoragename* and *targetstorage* arguments from Syntax 4.

Return value 6

Integer. Same as Syntax 5.

Usage

For Syntax 1 and 2, if you don't specify any arguments for SaveAs, PowerBuilder displays the Save Rows As dialog, letting the user specify the format of the saved data.



OLE storages

When you open an OLE object or call SaveAs for that object, PowerBuilder maintains a connection between the new storage and the object, so that future Save commands use the newly specified storage target.

The Open function establishes a connection between a storage file and a storage object, or a storage file or object and an OLE 2.0 control. The Save function uses this connection to save the OLE data.

When you call SaveAs for an OLE 2.0 control, it closes the current connection between the OLE object and its storage, either file or storage object. It establishes a new connection with the new storage, which will be the target of subsequent calls to the Save function.

When you call SaveAs for a storage object, it closes the current connection between the storage object and a file and creates a new file for the storage object's data.

For information about the structure of storage files, see the Open function.

Examples

This statement saves the contents of dw_History to the file G:\INVENTORY\EMPLOYEE.HIS. The saved file is in CSV format without column headings:

```
dw_History.SaveAs("G:\INVENTORY\EMPLOYEE.HIS",  &
   CSV!, FALSE)
```

Syntax 2

This statement saves the contents of the graph gr_History. The file and format information are not specified, so PowerBuilder will prompt for the file name and save the graph as tab-delimited text:

```
gr_History.SaveAs( )
```

This statement saves the contents of gr_History to the file G:\HR\EMPLOYEE.HIS. The format is CSV without column headings:

```
gr_History.SaveAs("G:\HR\EMPLOYEE.HIS" ,CSV!, FALSE)
```

This statement saves the contents of gr_computers in the DataWindow control dw_equipmt to the file G:\INVENTORY\SALES.XLS. The format is Excel with column headings:

```
dw_equipmt.SaveAs("gr_computers", &
     "G:\INVENTORY\SALES.XLS", Excel!, TRUE)
```

Syntax 3

This example saves the object in the control ole_1:

```
integer result
result = ole_1.SaveAs("c:\ole2\expense.ole")
```

Syntax 4

This example saves the object in the control ole_1 in the storage variable stg_stuff:

```
integer result
result = ole_1.SaveAs(stg_stuff)
```

Syntax 5

This example saves the storage object stg_stuff to the file MYSTUFF.OLE. Olest_stuff is an instance variable:

```
integer result
result = stg_stuff.SaveAs("c:\ole2\mystuff.ole")
```

This example opens a substorage in one file and saves it in another file. An OLE 2.0 storage file called MYROOT.OLE contains several substorages; one is called sub1. To open sub1 and save it in another file, the example defines two storage objects: stg1 and stg2. First MYROOT.OLE is opened into stg1. Next, sub1 is opened into stg2. Finally, stg2 is saved to the new file MYSUB.OLE. Just as when you open a word processing document and save it to a new name, the open object in stg2 is no longer associated with MYROOT.OLE; it is now connected to MYSUB.OLE:

```
olestorage stg1, stg2
stg1 = CREATE OLEStorage
stg2 = CREATE OLEStorage

stg1.Open("myroot.ole")
stg2.Open("sub1", stg1)

stg2.SaveAs("mysub.ole")
```

Syntax 6

This example saves the object in the OLEStorage variable stg_stuff in a second storage variable stg_clone as the substorage copy1:

```
integer result
result = stg_stuff.SaveAs("copy1", stg_clone)
```

See also

Close
Open
Print
Save
Update

Scroll

Description

Scrolls a MultiLineEdit control or the edit control of a DataWindow a specified number of lines up or down.

Applies to

DataWindow and MultiLineEdit controls

Syntax

editname.**Scroll** (*number*)

Parameter	Description
editname	The name of the DataWindow or MultiLineEdit in which you want to scroll up or down. If *editname* is a DataWindow, then Scroll affects its edit control.
number	The direction and number of lines you want to scroll. To scroll down, use a positive integer. To scroll up, use a negative integer.

Return value

Integer. Returns the line number of the top line in *editname* if it succeeds and *-1* if an error occurs.

Usage

If the number of lines left in the list is less than the number of lines that you want to scroll, then Scroll will scroll to the beginning or end, depending on the direction specified.

Examples

This statement scrolls mle_Employee down 4 lines:

```
mle_Employee.Scroll(4)
```

This statement scrolls mle_Employee up 4 lines:

```
mle_Employee.Scroll(-4)
```

See also

The following functions implement scrolling in a DataWindow:

ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollPriorRow
ScrollToRow

ScrollNextPage

Description

Scrolls a DataWindow control forward one page, displaying the next group of rows in the DataWindow's display area. (A page is the number of rows that can be displayed in the DataWindow control at one time.) ScrollNextPage changes the current row, but not the current column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ScrollNextPage** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you want to page (scroll) forward

Return value

Long. Returns the number of the row displayed at the top of the DataWindow control when the scroll is complete or if it tries to scroll past the last row. ScrollNextPage returns *-1* if an error occurs.

Usage

ScrollNextPage does not highlight the current row. Use SelectRow to let the user know what row is current.

For an example that uses RowCount and Describe to check whether the user has scrolled to the last page, see RowCount.

Events

ScrollNextPage may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Example

This statement scrolls dw_employee forward one page:

```
dw_employee.ScrollNextPage( )
```

See also

Scroll
ScrollNextRow
ScrollPriorPage
ScrollPriorRow

ScrollToRow
SelectRow

ScrollNextRow

Description

Scrolls a DataWindow control to the next row (forward one row). ScrollNextRow changes the current row, but not the current column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ScrollNextRow** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you want to scroll to the next row

Return value

Long. Returns the number of the row displayed at the top of the DataWindow control when the scroll is complete or if it tries to scroll past the last row. ScrollNextRow returns *-1* if an error occurs.

Usage

After you call ScrollNextRow, the row after the current row becomes the new current row. If that row is already visible, the displayed rows do not change. If it is not visible, the displayed rows move up to display the row.

ScrollNextRow does not highlight the row. Use SelectRow to let the user know what row is current.

Events

ScrollNextRow may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Example

This statement scrolls dw_employee to the next row:

```
dw_employee.ScrollNextRow( )
```

See also

Scroll
ScrollNextPage
ScrollPriorPage
ScrollPriorRow
ScrollToRow
SelectRow

ScrollPriorPage

Description

Scrolls a DataWindow control backward one page, displaying another group of rows in the DataWindow's display area. (A page is the number of rows that can be displayed in the DataWindow control at one time.) ScrollPriorPage changes the current row but not the current column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ScrollPriorPage** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you want to page (scroll) to the prior page

Return value

Long. Returns the number of the row displayed at the top of the DataWindow control when the scroll is complete or if it tries to scroll past the first row. ScrollPriorPage returns *-1* if an error occurs.

Usage

ScrollPriorPage does not highlight the current row. Use SelectRow to let the user know what row is current.

Events

ScrollPriorPage may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Example

This statement scrolls dw_employee backward one page:

```
dw_employee.ScrollPriorPage( )
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorRow
ScrollToRow
SelectRow

ScrollPriorRow

Description

Scrolls a DataWindow control backward one row. ScrollPriorRow changes the current row but not the current column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ScrollPriorRow** ( )

Parameter	Description
datawindowname	The name of the DataWindow control you want to scroll backward one row

Return value

Long. Returns the number of the row displayed at the top of the DataWindow control when the scroll is complete or if it tries to scroll past the first row. ScrollPriorRow returns *-1* if an error occurs.

Usage

After you call ScrollPriorRow, the row before the current row becomes the new current row. If that row is already visible, the displayed rows do not change. If it is not visible, the displayed rows move down to display the row.

ScrollPriorRow does not highlight the row. Use SelectRow to let the user know what row is current.

Events

ScrollPriorRow may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Example

This statement scrolls dw_employee to the prior row:

```
dw_employee.ScrollPriorRow( )
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollToRow
SelectRow

ScrollToRow

Description

Scrolls a DataWindow control to the specified row. ScrollToRow changes the current row but not the current column.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ScrollToRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow you want to scroll to a specific row.
row	A long identifying the row to which you want to scroll. If *row* is 0, ScrollToRow scrolls to the first row. If *row* is greater than the last row number, it scrolls to the last row.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

After you call ScrollToRow, the specified row becomes the new current row. If that row is already visible, the displayed rows do not change. If it is not visible, the displayed rows change to display the row.

ScrollToRow does not highlight the row. Use SelectRow to let the user know what row is current.

Events

ScrollToRow may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Example

This statement scrolls to row 10 and makes it current in the DataWindow control dw_employee:

```
dw_Employee.ScrollToRow(10)
```

See also

Scroll
ScrollNextPage
ScrollNextRow
ScrollPriorPage
ScrollPriorRow
SelectRow

Second

Description

Obtains the number of seconds in the seconds portion of a time value.

Syntax

Second (*time*)

Parameter	Description
time	The time value from which you want the seconds

Return value

Integer. Returns the seconds portion of *time* (00 to 59).

Examples

This statement returns 31:

```
Second(19:01:31)
```

See also

Hour
Minute
Second in Chapter 2, "DataWindow Painter Functions"

SecondsAfter

Description

Determines the number of seconds one time occurs after another.

Syntax

SecondsAfter (*time1*, *time2*)

Parameter	Description
time1	A time value that is the start time of the interval being measured
time2	A time value that is the end time of the interval

Return value

Long. Returns the number of seconds *time2* occurs after *time1*. If *time2* occurs before *time1*, SecondsAfter returns a negative number.

Examples

This statement returns 15:

```
SecondsAfter(21:15:30, 21:15:45)
```

This statement returns -15:

```
SecondsAfter(21:15:45, 21:15:30)
```

This statement returns 0:

```
SecondsAfter(21:15:45, 21:15:45)
```

If you declare start_time and end_time time variables and assign 19:02:16 to start_time and 19:02:28 to end_time as shown below:

```
time start_time, end_time
start_time = 19:02:16
end_time = 19:02:28
```

then each of these statements returns 12:

```
SecondsAfter(start_time, end_time)
SecondsAfter(19:02:16, end_time)
SecondsAfter(start_time, 19:02:28)
SecondsAfter(19:02:16, 19:02:28)
```

See also

DaysAfter
SecondsAfter in Chapter 2, "DataWindow Painter Functions"

Seek

Description

Moves the read/write pointer to the specified position in an OLE stream object. The pointer is the position in the stream at which the next read or write begins.

Applies to

OLEStream objects

Syntax

olestream.**Seek** (*position* {, *origin* })

Parameter	Description
olestream	The name of an OLE stream variable that has been opened.
position	A long whose value is the position relative to *origin* to which you want to move the read/write pointer.
origin (optional)	The value of the SeekType enumerated data type specifying where you want to start the seek. Values are: ♦ FromBeginning! — (Default) At the beginning of the file ♦ FromCurrent! — At the current position ♦ FromEnd! — At the end of the file

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

♦ *-1* Stream is not open

- *-2* Seek error
- *-9* Other error

Example

This example writes additional data to an OLE stream. First, it opens an OLE object in the file MYSTUFF.OLE and assigns it to the OLEStorage object stg_stuff. Then it opens the stream called info in stg_stuff and assigns it to the stream object olestr_info. Seek positions the read/write pointer at the end of the stream so that the contents of the instance blob variable lb_info is written at the end.

The example doesn't check the functions' return values for success, but you should be sure to check the return values in your code:

```
boolean lb_memexists
OLEStorage stg_stuff
OLEStream olestr_info

stg_stuff = CREATE OLEStorage
stg_stuff.Open("c:\ole2\mystuff.ole")

olestr_info.Open(stg_stuff, "info",  &
      stgReadWrite!, stgExclusive!)
olestr_info.Seek(0, FromEnd!)
olestr_info.Write(lb_info)
```

See also

Open
Length
Read
Write

SelectedIndex

Description

Obtains the number of the selected item in a ListBox control.

Applies to

ListBox controls

Syntax

listboxname.**SelectedIndex** ( )

Parameter	Description
listboxname	The name of the ListBox in which you want to locate the selected item

Return value

Integer. Returns the index of the selected item in *listboxname*. If more than one item is selected, SelectedIndex returns the index of the first selected item. If there are no selected items or an error occurs, SelectedIndex returns *-1*.

Usage

SelectedIndex and SelectedItem are meant for lists that allow a single selection only (the ListBox's MultiSelect attribute is FALSE).

When a ListBox's MultiSelect attribute is TRUE, SelectedIndex gets the index of the first selected item only. Use the State function, instead of SelectedIndex, to check each item in the list and find out if it is selected. Use the Text function to get the text of any item in the list.

Examples

If item 5 in lb_actions is selected, the this example sets Index to 5:

```
integer li_Index
li_Index = lb_actions.SelectedIndex( )
```

These statements open the window w_emp if item 5 in lb_actions is selected:

```
integer li_X
li_X = lb_actions.SelectedIndex( )
If li_X = 5 then Open(w_emp)
```

See also

SelectedItem

SelectedItem

Description

Obtains the text of the selected item in a ListBox control.

Applies to

ListBox controls

Syntax

listboxname.**SelectedItem** ( )

Parameter	Description
listboxname	The name of the ListBox in which you want the text of the currently selected item

Return value

String. Returns the text of the selected item in *listboxname*. Returns the empty string ("") if no items are selected.

Usage

SelectedItem and SelectItem are meant for lists that allow a single selection only (the ListBox's MultiSelect attribute is FALSE).

When a ListBox's MultiSelect attribute is TRUE, SelectedItem gets the text of the first selected item only. Use the State function, instead of SelectedItem, to check each item in the list and find out if it is selected. Use the Text function to get the text of any item in the list.

Example

If the text of the selected item in the ListBox lb_shortcuts is F1, then this example sets ls_item to F1:

```
string ls_Item
ls_Item = lb_Shortcuts.SelectedItem( )
```

See also

SelectedIndex
State

SelectedLength

Description

Determines the total number of characters in the selected text in an editable control, including spaces and carriage returns.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, and DropDownListBox controls

Syntax

editname.**SelectedLength** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox in which you want the length of the selected text. For a DataWindow, it reports the length of the selected text in the edit control over the current row and column.

Return value

Integer. Returns the length of the selected text in *editname*. If no text is selected, SelectedLength returns *0*. If an error occurs, SelectedLength returns *-1*.

Usage

A hard carriage return, produced by typing CTRL+ENTER, is a carriage return plus a line feed. It equals two characters when calculating the length.

For DropDownListBox controls, SelectedLength returns -1 if the control's AllowEdit attribute is set to FALSE.

> **Focus and the selection in a DropDownListBox**
> When a DropDownListBox loses focus, the selected text is no longer selected.

Examples

If the selected text in the MultiLineEdit mle_Contact is John Smith, then this example sets li_length to 10:

```
integer li_length
li_length = mle_Contact.SelectedLength( )
```

See also

LineLength
SelectedItem
SelectedLine
SelectedStart
TextLine

SelectedLine

Description

Obtains the number of the line that contains the insertion point in an editable control.

Applies to

DataWindow and MultiLineEdit controls

Syntax

editname.**SelectedLine** ( )

Parameter	Description
editname	The name of the DataWindow control or MultiLineEdit in which you want the number of the line containing the insertion point. For a DataWindow, it reports the line number in the edit control over the current row and column.

Return value

Integer. Returns the number of the cursor line in *editname*. If an error occurs, SelectedLine returns *-1*.

Usage

For EditMask controls, SelectedLine will compile but always returns 1.

Examples

If the cursor is positioned anywhere in line 5 of the MultiLineEdit mle_Contact, the following example sets li_SL to 5:

```
integer li_SL
li_SL = mle_Contact.SelectedLine( )
```

In this example, the line the user selects in the MultiLineEdit mle_winselect determines which window to open:

```
integer li_SL
li_SL = mle_winselect.SelectedLine( )

IF li_SL = 1 THEN
   Open(w_emp_data)
ELSEIF li_SL = 2 THEN
   Open(w_dept_data)
END IF
```

See also

LineLength
Position
SelectedText
TextLine

SelectedStart

Description

Reports the position of the first selected character in an editable control. The count begins at the start of the text and includes spaces and carriage returns.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, and DropDownListBox controls

Syntax

editname.**SelectedStart** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox in which you want to determine the starting position of selected text. For a DataWindow, it reports the starting position in the edit control over the current row and column.

Return value

Integer. Returns the starting position of the selected text in *editname*. If no text is selected, SelectedStart returns the position of the character immediately following the cursor. If an error occurs, SelectedStart returns *-1*.

Focus and the selection in a DropDownListBox
When a DropDownListBox loses focus, the selected text is no longer selected.

Example

If the MultiLineEdit mle_Comment contains Closed for Vacation July 3 to July 10, and Vacation is selected, then this example sets li_Start to 12 (the position of the first character in Vacation):

```
integer li_Start
li_Start = mle_Comment.SelectedStart( )
```

See also

Position
SelectedLength
SelectedLine

SelectedText

Description

Obtains the selected text in an editable control.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, and DropDownListBox controls

Syntax

editname.**SelectedText** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox from which you want the selected text. If the control is a DropDownListBox, the AllowEdit attribute must be TRUE. For a DataWindow, it reports the selected text in the edit control over the current row and column.

Return value

String. Returns the selected text in *editname*. If there is no selected text or if an error occurs, SelectedText returns the empty string ("").

> **Focus and the selection in a DropDownListBox**
> When a DropDownListBox loses focus, the selected text is no longer selected.

Examples

If the text in the MultiLineEdit mle_Contact is James B. Smith and James B. is selected, these statements set the value of emp_fname to James B.:

```
string ls_emp_fname
ls_emp_fname = mle_Contact.SelectedText( )
```

If the selected text in the edit portion of the DropDownListBox ddlb_Location is Maine, these statements display the ListBox lb_LBMaine:

```
string ls_Loc
ls_Loc = ddlb_Location.SelectedText( )
IF ls_Loc = "Maine" THEN
   lb_LBMaine.Show()
ELSE
   . . .
END IF
```

SelectItem

Description

Finds and highlights an item in a ListBox or DropDownListBox control. There are two syntaxes:

- When you know the text of the item, but not its position, use Syntax 1.
- When you know the position of the item in the control's list or you want to clear the current selection, use Syntax 2.

Applies to

ListBox and DropDownListBox controls

Syntax 1

listboxname.**SelectItem** (*item*, *index*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want to select a line
item	A string whose value is the starting text of the item you want to select
index	The number of the item after which you want to begin the search

Return value 1

Integer. Returns the index number of the selected item. If no match is found, SelectItem returns *0*; it returns *-1* if an error occurs.

Syntax 2

listboxname.**SelectItem** (*itemnumber*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want to select an item.
itemnumber	An integer whose value is the location (index) of the item in the ListBox or the listbox portion of the DropDownListBox. Specify 0 for *itemnumber* to deselect the selected item. For a ListBox, 0 removes highlighting from the selected item. For a DropDownListBox, 0 clears the edit box.

Return value 2

Integer. Returns the index number of the selected item. SelectItem returns *0* if *itemnumber* is not valid or if you specified 0 in order to deselect the selected item. It returns *-1* if an error occurs.

Usage

Syntax 1 of SelectItem begins searching for the desired item after the item identified by *index*. To match, the item must start with the specified text; however, the text in the item can be longer than the specified text.

To find an item but not select it, use the FindItem function.

MultiSelect ListBoxes
SelectItem has no effect on a ListBox whose MultiSelect attribute is TRUE. Instead, use SetState to select items without affecting the selected state of other items in the list.

Clearing the edit box of a DropDownListBox
To clear the edit box of a DropDownListBox that the user cannot edit, use Syntax 2 of SelectItem and set *itemnumber* to 0. Setting the control's text to the empty string doesn't work if the control's AllowEdit attribute is FALSE.

Examples

If item 5 in lb_Actions is Delete Files, this example starts searching after item 2, finds and highlights Delete Files, and sets li_Index to 5:

```
integer li_Index
li_Index = lb_Actions.SelectItem("Delete Files", 2)
```

If item 4 in lb_Actions is Select Objects, this example starts searching after item 2, finds and highlights Select Objects, and sets li_Index to 4:

```
integer li_Index
li_Index = lb_Actions.SelectItem("Sel", 2)
```

Syntax 2

This example highlights item number 5:

```
integer li_Index
li_Index = lb_Actions.SelectItem(5)
```

This example clears the selection from the edit box of the DropDownListBox ddlb_choices and sets li_Index to 0:

```
integer li_Index
li_Index = ddlb_choices.SelectItem(0)
```

See also

AddItem
DeleteItem
FindItem
InsertItem
SetState

SelectObject

Description

Selects or deselects the object in an OLE 2.0 control and, when selecting, activates the server application. The server's menus are added to the PowerBuilder application's menus.

Applies to

OLE 2.0 controls

Syntax

ole2control.**SelectObject** (*selectstate*)

Parameter	Description
ole2control	The name of the OLE 2.0 control containing the object you want to select.
selectstate	A boolean value indicating whether you want to select or deselect the object.

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Control is empty
- *-9* Other error

Example

This example selects the object in the OLE 2.0 control ole_1:

```
integer result
result = ole_1.SelectObject(TRUE)
```

SelectRow

Description

Highlights or unhighlights rows in a DataWindow control. You can select all rows or a single row. SelectRow does not affect which row is current. It does not select rows in the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SelectRow** (*row, boolean*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to select or deselect a row.
row	A long identifying the row you want to select or deselect. Specify 0 to select or deselect all rows.
boolean	A boolean value that determines whether the row is selected or not selected: ♦ TRUE — Select the row(s) ♦ FALSE — Deselect the row(s)

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If a row is already selected and you specify that it be selected (*boolean* is TRUE), it remains selected. If a row is not selected and you specify that it not be selected (*boolean* is FALSE), it remains unselected.

Examples

This statement selects the 15 row in dw_employee:

```
dw_employee.SelectRow(15, TRUE)
```

As the script for a DataWindow's Clicked event, this example removes highlighting from all rows and then highlights the row the user clicked:

```
This.SelectRow(0, FALSE)
This.SelectRow(This.GetClickedRow(), TRUE)
```

SelectText

Description

Selects text in an editable control. You specify where the selection begins and how many characters to select.

Applies to

DataWindow, EditMask, MultiLineEdit, SingleLineEdit, and DropDownListBox controls

Syntax

editname.**SelectText** (*start*, *length*)

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, SingleLineEdit, or DropDownListBox in which you want to select text.
start	The position at which you want to start the selection.
length	The number of characters you want to select. If *length* is 0, no text is selected but PowerBuilder moves the insertion point to the location specified in *start*.

Return value

Integer. Returns the number of characters selected. If an error occurs, SelectText returns *-1*.

Usage

If the control does not have the focus when you call SelectText, then the text is not highlighted until the control has focus. To set focus on the control so that the selected text is highlighted, call the SetFocus function.

> **Tip**
> When you want to select all the text of a line edit or select the contents from a specified position to the end of the edit, use the Len function to obtain the length of the control's text.

Examples

This statement sets the insertion point at the end of the text in the SingleLineEdit sle_name:

```
sle_name.SelectText(Len(sle_name.Text), 0)
```

This statement selects the entire contents of the SingleLineEdit sle_name:

```
sle_name.SelectText(1, Len(sle_name.Text))
```

The rest of these examples assume the MultiLineEdit mle_EmpAddress contains Boston Street.

The following statement selects the string ost and returns 3:

```
mle_EmpAddress.SelectText(2, 3)
```

The next statement selects the string oston Street and returns 12:

```
mle_EmpAddress.SelectText(2,  &
   Len(mle_EmpAddress.Text))
```

These statements select the string Bos, return 3, and sets the focus to mle_EmpAddress so that Bos is highlighted:

```
mle_EmpAddress.SelectText(1, 3)
mle_EmpAddress.SetFocus()
```

See also

Len
Position
SelectedItem
SelectedText
SetFocus
TextLine

Send

Description

Sends a message to a window so that it is executed immediately.

Syntax

Send (*handle*, *message#*, *lowword*, *long*)

Parameter	Description
handle	A long whose value is the system handle of a window (that you have created in PowerBuilder or another application) to which you want to send a message.
message#	An UnsignedInteger whose value is the system message number of the message you want to send.
word	A long whose value is the integer value of the message. If this argument is not used by the message, enter 0.
long	The long value of the message or a string.

Return value

Long. Returns the value returned by SendMessage in Windows if it succeeds and *-1* if an error occurs.

Usage

PowerBuilder's Send function sends the message identified by *message#* and optionally, *lowword* and *long*, to the window identified by *handle* to the Windows function SendMessage. The message is send directly to the object, bypasses the object's message queue. Send waits until the message is processed and obtains the value returned by SendMessage.

Messages in Windows
Use the Handle function to get the Windows handle of a PowerBuilder object.

You specify Windows messages by number. They are documented in the file WINDOWS.H that is part of the Microsoft Windows Software Development Kit (SDK) and other Windows development tools.

Tip
Messages sent with Send are executed immediately. To post a message to the end of an object's message queue, use the Post function.

Examples

This statement scrolls the window w_emp up one page:

```
Send(Handle(w_emp), 277, 2, 0)
```

Both of the following statements click the CheckBox cb_OK:

```
Send(Handle(Parent), 273, 0, Long(cb_OK.Handle()))
cb_OK.TriggerEvent(Clicked!)
```

You can send messages to maximize or minimize a DataWindow, and return it to normal. To use these messages, enable the TitleBar, Minimize, and Maximize attributes of your DataWindow control. Also, you should give your DataWindow control an icon for its minimized state.

This statement minimizes the DataWindow:

```
Send(Handle(dw_whatever), 274, 61472, 0)
```

This statement maximizes the DataWindow:

```
Send(Handle(dw_whatever), 274, 61488, 0)
```

This statement returns the DataWindow to its normal, defined size:

```
Send(Handle(dw_whatever), 274, 61728, 0)
```

You can send a Windows message to determine the last item clicked in a multiselect ListBox. The following script for the SelectionChanged event of a ListBox control gets the return value of the LB_GETCURSEL message which is the item number in the list (where the first item is 0, not 1). To get PowerBuilder's index for the list item, the example adds 1 to the return value from Send. In this example, idx is an integer instance variable for the window:

```
// Send the Windows message for LB_GETCURSEL
// to the listbox
idx = Send(Handle(This), 1033, 0, 0)
idx = idx + 1
```

See also

Handle
Post

SeriesCount

Description

Counts the number of series in a graph.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**SeriesCount** ({ *graphcontrol* })

Parameter	Description
controlname	The name of the graph for which you want the number of series, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the number of series

Return value

Integer. Returns the number of series in the graph if it succeeds and *-1* if an error occurs.

Examples

These statements store in the variable li_series_count the number of series in the graph gr_product_data:

```
integer li_series_count
li_series_count = gr_product_data.SeriesCount( )
```

These statements store in the variable li_series_count the number of series in the graph gr_computers in the DataWindow control dw_equipment:

```
integer li_series_count
li_series_count = &
   dw_equipment.SeriesCount("gr_computers")
```

See also

CategoryCount
DataCount

SeriesName

Description

Obtains the series name associated with the specified series number.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**SeriesName** ({ *graphcontrol*, } *seriesnumber*)

Parameter	Description
controlname	The name of the graph in which you want the name of a series, or the name of the DataWindow control containing the graph
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want the name of a series
seriesnumber	The number of the series for which you want to obtain the name

Return value

String. Returns the name assigned to the series. If an error occurs, it returns the empty string ("").

Usage

Series are numbered consecutively, from 1 to the value returned by SeriesCount. When you delete a series, the series are renumbered to keep the numbering consecutive. You can use SeriesName to find out the name of the series associated with a series number.

Examples

These statements store in the variable ls_SeriesName the name of series 5 in the graph gr_product_data:

```
string ls_SeriesName
ls_SeriesName = gr_product_data.SeriesName(5)
```

These statements store in the variable ls_SeriesName the name of series 5 in the graph gr_computers in the DataWindow control dw_equipment:

```
string ls_SeriesName
ls_SeriesName = &
   dw_equipment.SeriesName("gr_computers", 5)
```

See also

CategoryName
DeleteSeries
FindSeries

SetActionCode

Description

Sets the action code for an event in a DataWindow control. The action code determines the action that PowerBuilder takes following the event. The default action code is 0.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetActionCode** (*code*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set an action code.
code	A long whose value specifies the action you want to take in the DataWindow control. The meaning of the action code depends on the event.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use SetActionCode to change the action that occurs following a DataWindow event. Not all DataWindow events have action codes, only those events that can have different outcomes.

For a list of events and their action codes, see *Objects and Controls*.

> **Tip**
> Although SetActionCode is not required to be the last statement in a script, it should be the last statement. When it is not the last statement it may not perform as expected.

Example

In the ItemChanged event script for dw_Employee, these statements set the action code in dw_Employee to reject data that is less than the employee's age:

```
integer a, age
age = Integer(sle_Age.Text)
a = Integer(dw_Employee.GetText( ))
IF a < age THEN dw_Employee.SetActionCode(1)
```

This example shows a script for the DBError event script that displays a version of the error message to the user. Because PowerBuilder also displays a message to the user after the event, the script calls SetActionCode to set the action code to 1, which suppresses PowerBuilder's error message:

```
integer errnum
errnum = dw_emp.DBErrorCode()

// Show error code and message to the user
MessageBox("Database Error",  &
   "Number " + String(errnum) + " " +  &
   dw_emp.DBErrorMessage(), StopSign!)

// Stop PowerBuilder from displaying its message
dw_emp.SetActionCode(1)
```

SetBorderStyle

Description

Sets the border style of a column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetBorderStyle** (*column*, *borderstyle*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to change the style of a column border.
column	The column in which you want to change the border style. *Column* can be a column number (integer) or a column name (string).
borderstyle	A value of the Border enumerated data type indicating the border style you want to use for the column. Values are: ♦ Box! ♦ NoBorder! ♦ ShadowBox! ♦ Underline!

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Example

This example checks the border of column 2 in dw_emp and, if there is no border, gives it a shadow box border:

```
Border B3
B3 = dw_emp.GetBorderStyle(2)
IF B3 = NoBorder! THEN  &
   dw_emp.SetBorderStyle(2, ShadowBox!)
```

See also

GetBorderStyle

SetColumn

Description

Sets the current column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetColumn** (*column*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the current column.
column	The column you want to make current. *Column* can be a column number (integer) or a column name (string).

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. If *column* is less than 1 or greater than the number of columns, SetColumn fails.

Usage

SetColumn moves the cursor to the current column but does not scroll the DataWindow control.

Only an editable column can be current. (A column is editable when it tab order value is greater than 0.) Do not try to make a column that can't be edited current.

Events

SetColumn may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged

Avoiding infinite loops
Never call SetColumn in the ItemChanged, ItemError, or ItemFocusChanged event. Because SetColumn can trigger these events, such a recursive call can cause a stack fault.

Example

This statement makes the 15th column in dw_Employee the current column:

```
dw_Employee.SetColumn(15)
```

See also

GetColumn
GetRow
SetRow

SetDataDDE

Description

Sends data to a DDE client application when PowerBuilder is acting as a DDE server. You would usually call SetDataDDE in the script for the RemoteRequest event, which is triggered by a DDE request for data from the client application.

Platform information
This and other DDE functions have no effect on the Macintosh.

Syntax

SetDataDDE (*string* {, *applname*, *topic*, *item* })

Parameter	Description
string	The data you want to send to a DDE client application
applname (optional)	The DDE name for the client application
topic (optional)	A string whose value is the basic data grouping the DDE client application referenced
item (optional)	A string (data within *topic*)

Return value

Integer. Returns *1* if it succeeds. If an error occurs, SetDataDDE returns a negative integer. Values are:

- *-1* Function called in the wrong context
- *-2* Data not accepted

Usage

To enable DDE server mode in your PowerBuilder application, call the StartServerDDE function. Then DDE messages from a DDE client will trigger events in the PowerBuilder window. It is up to you to decide how your application responds by writing code for those events. When an application requests data of the DDE server, it triggers a RemoteRequest event. You typically call SetDataDDE in the script for a window's RemoteRequest event.

If a client application has established a hot link with a location in your PowerBuilder application, you can call SetDataDDE in an event for the object associated with the location. As a server application, you decide how location names map to the controls in your application. For example, your application can decide that the DDE name loc1 refers to the SingleLineEdit sle_name and a client application can establish a hot link with "loc1." Then in the Modified event for sle_name, you can call SetDataDDE so that the client application receives changes each time sle_name is changed. Likewise, if loc1 referred to a DataWindow, you can call SetDataDDE in the ItemChanged event for the DataWindow.

The *applname* argument refers to the client application that has established a channel or a hot link with your application. *Topic* and *item* refer to a topic and location recognized by your server application. You only need to specify these arguments to make it clear to the client application who should receive the message and what is being sent.

Examples

This statement illustrates how SetDataDDE is used in a script for a RemoteRequest event when another DDE application requests data. The data sent is the text of the SingleLineEdit sle_Address:

```
SetDataDDE(sle_Address.Text)
```

This statement illustrates how the optional arguments are specified:

```
SetDataDDE(sle_Address.Text, "MYDB",  &
   "Employee", "Address")
```

See also

GetDataDDE
StartServerDDE

SetDataPieExplode

Description

Explodes a pie slice in a pie graph. The exploded slice is moved away from the center of the pie, which draws attention to the data. You can explode any number of slices of the pie.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax

controlname.**SetDataPieExplode (** { *graphcontrol*, } *series*, & *datapoint*, *percentage* **)**

Parameter	Description
controlname	The name of the graph in which you want to explode a pie slice, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to explode a pie slice.
seriesnumber	The number that identifies the series.
datapoint	The number of the data point (that is, the pie slice) to be exploded.
percentage	A number between 0 and 100 which is the percentage of the radius that the pie slice is moved away from the center. When *percentage* is 100, the tip of the slice is even with the circumference of the pie's circle.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If the graph is not a pie graph, the function has no effect.

Example

This example explodes the pie slice under the pointer to 50% when the user double-clicks within the graph. The code checks the attribute GraphType to make sure the graph is a pie graph. It then finds out whether the user clicked on a pie slice by checking the series and data point values set by ObjectAtPointer. The script is for the DoubleClicked event of a graph object:

```
integer series, datapoint
grObjectType  clickedtype
integer percentage

percentage = 50
IF (This.GraphType <> PieGraph! AND  &
   This.GraphType <> Pie3D!) THEN RETURN
clickedtype = This.ObjectAtPointer(  &
   series, datapoint)

IF (series > 0 and datapoint > 0) THEN
   This.SetDataPieExplode(series, datapoint,  &
      percentage)
END IF
```

See also GetDataPieExplode

SetDataStyle

Description Specifies the appearance of a data point in a graph. The data point's series has appearance settings that you can override with SetDataStyle. There are several syntaxes:

- To set the data point's colors, use Syntax 1.
- To set the line style and width for the data point, use Syntax 2.
- To set the fill pattern or symbol for the data point, use Syntax 3.

Applies to Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax 1 *controlname*.**SetDataStyle** ({ *graphcontrol*, } *seriesnumber*, & *datapointnumber*, *colortype*, *color*)

Parameter	Description
controlname	The name of the graph in which you want to set the color of a data point, or the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the color of a data point.
seriesnumber	The number of the series in which you want to set the color of a data point.
datapointnumber	The number of the data point for which you want to set the color.

Parameter	Description
colortype	A value of the grColorType enumerated data type specifying the aspect of the data point for which you want to set the color. Values are: ♦ Foreground! — Text color ♦ Background! — Background color ♦ LineColor! — Line color ♦ Shade! — Shade (for graphics that are three-dimensional or have solid objects)
color	A long whose value is the new color for *colortype*

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

controlname.**SetDataStyle** ({ *graphcontrol*, } & *seriesnumber*, *datapointnumber*, *linestyle*, *linewidth*)

Parameter	Description
controlname	The name of the graph in which you want to set the line style and width of a data point, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the line style and width.
seriesnumber	The number of the series in which you want to set the line style and width of a data point.
datapointnumber	The number of the data point for which you want to set the line style and width.
linestyle	A value of the LineStyle enumerated data type. Values are: ♦ Continuous! ♦ Dash! ♦ DashDot! ♦ DashDotDot! ♦ Dot! ♦ Transparent!
linewidth	An integer whose value is the width of the line in pixels

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 3

controlname.**SetDataStyle** ({ *graphcontrol*, } & *seriesnumber*, *datapointnumber*, *enumvalue*)

Parameter	Description
controlname	The name of the graph in which you want to set the appearance of a data point, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the appearance.
seriesnumber	The number of the series in which you want to set the appearance of a data point.
datapointnumber	The number of the data point for which you want to set the appearance.
enumvalue	An enumerated data type specifying the appearance setting for the data point. You can specify a FillPattern or grSymbolType value. To change the fill pattern, use a FillPattern value: ♦ Bdiagonal! — Lines from lower left to upper right ♦ Diamond! ♦ Fdiagonal! — Lines from upper left to lower right ♦ Horizontal! ♦ Solid! ♦ Square! ♦ Vertical! To change the symbol type, use a grSymbolType value: ♦ NoSymbol! ♦ SymbolHollowBox! ♦ SymbolX! ♦ SymbolStar! ♦ SymbolHollowUpArrow! ♦ SymbolHollowCircle! ♦ SymbolHollowDiamond! ♦ SymbolSolidDownArrow! ♦ SymbolSolidUpArrow!

Parameter	Description
	♦ SymbolSolidCircle! ♦ SymbolSolidDiamond! ♦ SymbolPlus! ♦ SymbolHollowDownArrow! ♦ SymbolSolidBox!

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

To change the appearance of a series, use SetSeriesStyle. The settings you make for the series are the defaults for all data points in the series.

To reset the color of individual points back to the series color, call ResetDataColors.

For a graph in a DataWindow, you can specify the appearance of a data point in the graph before PowerBuilder draws the graph. To do so, define a user event for pbm_dwngraphcreate and call SetDataStyle in the script for that event. The event pbm_dwngraphcreate is triggered just before a graph is created in a DataWindow object.

Examples

This example checks the background color for data point 6 in the series named Salary in the graph gr_emp_data. If it is red, this SetDataStyle sets it to black:

```
long color_nbr
integer SeriesNbr

// Get the number of the series
SeriesNbr = gr_emp_data.FindSeries("Salary")

// Get the background color
gr_emp_data.GetDataStyle(SeriesNbr, 6,  &
   Background!, color_nbr)

// If color is red, change it to black
IF color_nbr = 255 THEN  &
   gr_emp_data.SetDataStyle(SeriesNbr, 6,  &
      Background!, 0)
```

These statements set the text (foreground) color to black for data point 6 in the series named Salary in the graph gr_depts in the DataWindow control dw_employees:

```
integer SeriesNbr

// Get the number of the series
SeriesNbr = &
   dw_employees.FindSeries("gr_depts", "Salary")
```

```
// Set the background color
dw_employees.SetDataStyle("gr_depts", SeriesNbr,  &
   6, Background!, 0)
```

Syntax 2

This example checks the line style used for data point 10 in the series named Costs in the graph gr_computers in the DataWindow control dw_equipment. If it is dash-dot, the SetDataStyle sets it to continuous. The line width stays the same:

```
integer SeriesNbr, line_width
LineStyle line_style

// Get the number of the series
SeriesNbr = dw_equipment.FindSeries(  &
   "gr_computers", "Costs")

// Get the current line style
dw_equipment.GetDataStyle("gr_computers",  &
   SeriesNbr, 10, line_style, line_width)

// If the pattern is dash-dot, change to continuous
IF line_style = DashDot! THEN  &
   dw_equipment.SetDataStyle("gr_computers",  &
   SeriesNbr, 10, Continuous!, line_width)
```

Syntax 3

This example checks the fill pattern used for data point 10 in the series named Costs in the graph gr_product_data. If it is diamond, then SetDataStyle changes it to solid:

```
integer SeriesNbr
FillPattern data_pattern

// Get the number of the series
SeriesNbr = gr_product_data.FindSeries("Costs")

// Get the current fill pattern
gr_product_data.GetDataStyle(SeriesNbr, 10,  &
    data_pattern)

// If the pattern is diamond, change it to solid
IF data_pattern = Diamond! THEN  &
   gr_product_data.SetDataStyle(SeriesNbr,  &
   10, Solid!)
```

See also

GetDataStyle
ResetDataColors
SeriesName
GetSeriesStyle
SetSeriesStyle

SetDetailHeight

Description

Sets the height of each row in the specified range to the specified value.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetDetailHeight** (*startrow, endrow, height*)

Parameter	Description
datawindowname	The name of the DataWindow control for which you want to set the height of one or more rows in the detail area
startrow	A long whose value is the first row in the range of rows for which you want to set the height
endrow	A long whose value is the last row in the range of rows for which you want to set the height
height	The height of the detail area for the specified rows in the units specified for the DataWindow object

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Call SetDetailHeight in a script to vary the amount of space assigned to rows in a DataWindow control. You cannot specifically set the height for different rows when you define a DataWindow object in the DataWindow painter, although you can turn on the Autosize Height attribute for the detail band so that the height of each row is determined by the data.

You can set the detail height of one or more rows to zero, which hides them from view.

Examples

This statement sets the height of rows 2 and 3 to 500:

```
dw_1.SetDetailHeight(2, 3, 500)
```

This statement sets the height of rows 2 and 3 to 0, so the rows in the DropDownDataWindow do not display:

```
dw_child.SetDetailHeight(2, 3, 0)
```

This script retrieves rows for a DropDownDataWindow associated with the Company_Name column. It then hides rows 2 and 3 of the DropDownDataWindow:

```
DataWindowChild dwc
integer rtncode

rtncode = dw_1.dwGetChild("company_name", dwc)
IF rtncode < 0 THEN HALT
dwc.SetTransObject(SQLCA)
dwc.Retrieve( )
dwc.SetDetailHeight(2, 3, 0)
```

SetDynamicParm

Description

Specifies a value for an input parameter in the DynamicDescriptionArea that will be used in an SQL OPEN or EXECUTE statement.

> **Note**
> Use this function only in conjunction with Format 4 dynamic SQL statements.

Syntax

DynamicDescriptionArea.**SetDynamicParm** (*index*, *value*)

Parameter	Description
DynamicDescriptionArea	The name of the DynamicDescriptionArea, usually SQLDA.
index	An integer identifying the input parameter descriptor in which you want to set the data. *Index* must be less than or equal to the value in NumInputs in *DynamicDescriptionArea*.
value	The value you want to use to fill the input parameter descriptor identified by *index*.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

SetDynamicParm specifies a value for the parameter identified by *index* in the array of input parameter descriptors in *DynamicDescriptionArea*.

Use SetDynamicParm to fill the parameters in the input parameter descriptor array in the DynamicDescriptionArea before executing an OPEN or EXECUTE statement.

Examples

This statement fills the first input parameter descriptor in SQLDA with the string MA:

```
SQLDA.SetDynamicParm(1, "MA")
```

This statement fills the fourth input parameter descriptor in SQLDA with the number 01742:

```
SQLDA.SetDynamicParm(4, "01742")
```

This statement fills the third input parameter descriptor in SQLDA with the date 12-31-1992:

```
SQLDA.SetDynamicParm(3, "12-31-1992")
```

See also

GetDynamicDate
GetDynamicDateTime
GetDynamicNumber
GetDynamicString
GetDynamicTime
Discussion of dynamic SQL in *PowerScript Language*
OPEN (SQL command) in *PowerScript Language*

SetFilter

Description

Specifies filter criteria for a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetFilter** (*format*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to define the filter.
format	A string whose value is a boolean expression that you want to use as the filter criteria. The expression includes column names or numbers. A column number must be preceded by a pound sign (#). If *format* is NULL, PowerBuilder prompts you to enter a filter.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

A DataWindow object can have filter criteria specified as part of its definition. After data is retrieved, rows that don't meet the criteria are immediately transferred from the primary buffer to the filter buffer.

The SetFilter function replaces the filter criteria defined for the DataWindow object, if any, with a new set of criteria. To apply the filter criteria to the DataWindow control, call the Filter function, which transfers rows that do not meet the filter criteria to the filter buffer.

The filter expression consists of columns, relational operators, and values against which column values are compared. Boolean expressions can be connected with logical operators AND and OR. You can also use NOT, the negation operator. Use parentheses to control the order of evaluation.

Sample expressions are:

```
item_id > 5
NOT item_id = 5
(NOT item_id = 5) AND customer > "Mabson"
item_id > 5 AND customer = "Smith"
#1 > 5 AND #2 = "Smith"
```

The filter expression is a string and does not contain variables. However, you can build the string at during execution using the values of variables in the script. Within the filter string, values that are strings must be enclosed in quotation marks (see the examples).

If the filter expression contains numbers, the DataWindow expects the numbers in U.S. format. Be aware that the String function formats numbers using the current system settings. If you use it to build the filter expression, specify a display format that produces U.S. notation.

Removing a filter

To remove a filter, call SetFilter with the empty string ("") for *format* and then call Filter. The rows in the filter buffer will be restored to the primary buffer after the rows that were already in the primary buffer.

If you specify a null filter expression, rather than an empty string, PowerBuilder displays the Specify Filter dialog when you call Filter so users can specify their own filter expression (see Filter).

Examples

This statement defines the filter expression for dw_Employee as the value of format1:

```
dw_Employee.SetFilter(format1)
```

The following statements define a filter expression and set it as the filter for dw_Employee. With this filter, only those rows in which the cust_qty column exceeds 100 and the cust_code column exceeds 30 are displayed:

```
string DWfilter2
DWfilter2 = "cust_qty > 100 and cust_code >30"
dw_Employee.SetFilter(DWfilter2)
```

The following statements define a filter so that emp_state of dw_Employee displays only if it is equal to the value of var1 (in this case ME). The filter expression passed to SetFilter is emp_state = ME:

```
string Var1
Var1 = "ME"
dw_Employee.SetFilter("emp_state = '"+ var1 +" '")
```

The following statements define a filter so that column 1 must equal the value in min_qty and column 2 must equal the value in max_qty to pass the filter. The resulting filter expression is:

```
#1=100 and #2=1000
```

The sample code is:

```
integer max_qty, min_qty

min_qty = 100
max_qty = 1000

dw_inv.SetFilter(dw_1, "#1="+ string( min_qty)  &
   + " and #2=" + string(max_qty))
```

See also

Filter

SetFocus

Description

Sets the focus on the specified object or control.

Applies to

Any object

Syntax

objectname.**SetFocus ( )**

Parameter	Description
objectname	The name of the object or control in which you want to set the focus

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If *objectname* is a ListBox, SetFocus displays the focus rectangle around the first item. If *objectname* is a DropDownListBox, SetFocus highlights the edit box. To select an item in a ListBox or DropDownListBox, use SelectItem.

Drawing objects cannot have focus. Therefore, you cannot use SetFocus to set focus to in a Line, Oval, Rectangle, or RoundRectangle.

Example

This statement in the script for the Open event in a window moves the focus to the first item in lb_Actions:

```
lb_Actions.SetFocus( )
```

See also

SelectItem
SetState
SetTop

SetFormat

Description

Specifies a display format for a column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetFormat** (*column*, *format*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to specify the display format.
column	The column for which you are specifying the display format. *Column* can be a column number (integer) or a column name (string).
format	A string whose value is the display format for the DataWindow column.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

For information on valid display formats for different data types, see the *User's Guide* or the Display Formats dialog in the DataWindow painter.

If you are specifying the display format for a number, the format must use U.S. notation. That is, comma represents the thousands delimiter and period represents the decimal place. During execution, the locally correct symbols will be displayed.

Example

These statements define the display format for column 15 of dw_employee to the contents of format1:

```
string format1
format1 = "$#,##0.00"
dw_employee.SetFormat(15, format1)
```

See also

GetFormat

SetItem

Description

Sets the value of a row and column in a DataWindow control to the specified value.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetItem** (*row, column, value*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set a specific row and column to a value.
row	A long whose value is the row location of the data.
column	The column location of the data. *Column* can be a column number (integer) or a column name (string).
value	The value to which you want to set the data at the row and column location. The data type of the value must be the same data type as the column.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

SetItem sets a value in a DataWindow buffer. It does not affect the value currently in the edit control over the current row and column, which is the data the user has changed or may change. The value in the edit control does not become the value of the DataWindow item until it is validated and accepted (see AcceptText). In a script, you can change the value in the edit control with the SetText function.

You can use SetItem when you want to the set the value of an item in a DataWindow control that has script as the source.

You can also use SetItem to set the value of an item when the data the user entered is not valid. When you set an action code that rejects the data the user entered but allows the focus to change (action code of 2 in the script of the ItemChanged event or action code of 3 in the ItemError event), you can call SetItem to put valid data in the row and column.

Tip

If PowerBuilder cannot convert the string the user entered properly, you will have to include statements in the script for the ItemChanged or ItemError event to convert the data and use SetItem with the converted data. For example, if the user enters a number with commas and a dollar sign (for example, $1,000), PowerBuilder is unable to convert the string to a number and you will have to convert it in the script.

If you use SetItem to set a row and column to a value other than the value the user entered, you can use SetText to assign the new value to the edit control so that the user sees the current value.

Examples

This statement sets the value of row 3 of the column named hire_date of the DataWindow control dw_order to 1993-06-07:

```
dw_order.SetItem(3, "hire_date", 1993-06-07)
```

When a user starts to edit a numeric column and leaves it without entering any data, PowerBuilder tries to assign an empty string to the column. This fails the data type validation test. In this example, code in the ItemError event sets the column's value to NULL and allows the focus to change.

This example assumes that the data type of column 2 is numeric. If it is date, time, or datetime, replace the first line (integer null_num) with a declaration of the appropriate data type.

```
integer null_num     //to contain null value
integer col_no

SetNull(null_num)
col_no = This.GetColumn()

// Special processing for column 2
IF col_no = 2 THEN
   // If user entered nothing (""), set to null
   IF This.GetText() = "" THEN
      This.SetItem(This.GetRow(), col_no, null_num)
      This.SetActionCode(2)
      RETURN
   END IF
END IF
```

The following example is a script for a DataWindow's ItemError event. If the user specifies characters other than digits for a numeric column, the data will fail the data type validation test. You can include code to strip out characters such as commas and dollar signs and use SetItem to assign the now valid numeric value to the column. The action code of 3 causes the data in the edit control to be rejected because the script has provided a valid value:

```
string s, snum, c
integer cnt

s = This.GetText()

// Extract the digits from the user's data
FOR cnt = 1 to Len(s)
   c = Mid(s, cnt, 1)
   IF IsNumber(c) THEN snum = snum + c
NEXT

This.SetItem(This.GetRow(), This.GetColumn(),  &
   Long(snum))
This.SetActionCode(3)
```

See also

GetItemDate
GetItemDateTime
GetItemNumber
GetItemString
GetItemTime
GetText
SetActionCode
SetText

SetItemStatus

Description

Changes the modification status of a row or a column within a row. The modification status determines the type of SQL statement the Update function will generate for the row.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetItemStatus** (*row, column, dwbuffer, status*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the status of an item at a specified row and column.
row	A long identifying the row location in which you want to set the status.
column	The column location in which you want to set the status. *Column* can be a column number (integer) or a column name (string). To set the status for the row, enter 0 for *column*.
dwbuffer	A dwBuffer enumerated data type identifying the DataWindow buffer that contains the row: ♦ PRIMARY! — The data in the primary buffer (the data that has not been deleted or filtered out) ♦ DELETE! — The data in the delete buffer (data deleted from the DataWindow object) ♦ FILTER! — The data in the filter buffer (data that was filtered out)
status	A dwItemStatus enumerated data type representing the new status: ♦ NotModified! ♦ DataModified! ♦ New! ♦ NewModified! For definitions of the values of dwItemStatus, see GetItemStatus.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A row's status flag determines what SQL command the Update function uses to update the database. Changing a row's status flag affects the status flags of the row's columns and vice versa. For example, changing a row's status to NotModified! or New! will change the columns in that row to NotModified! as well.

For rows, not all status changes are valid. For example, you cannot change NewModified! to New!. Some status changes, although allowed, result in a different status than you specify. For example, changing DataModified! to New! results in a status of NewModified!.

The following table illustrates the effect of changing the row's original status to another status specified with SetItemStatus. If the table says Yes, then the specified status takes effect. If the table says No, specifying that status in SetItemStatus has no effect. If the table specifies a different status, it is the status that results from the status you specify.

	Specified Status			
Original Status	New!	NewModified!	DataModified!	NotModified!
New!	–	Yes	Yes	No
NewModified!	No	–	Yes	New!
DataModified!	NewModified!	Yes	–	Yes
NotModified!	Yes	Yes	Yes	–

When a particular status change is not allowed, you can call SetItemStatus more than once to set the row to the desired setting. For example, if you want to set a row with New! status to NotModified!, you can set it first to DataModified! and then to NotModified!.

Use SetItemStatus when you want to change the way a row will be updated. For example, if you copy a row from one DataWindow to another and the user modifies the row, that row will have a status of NewModified!. However, the row already exists in the database, so you want the Update to function to use the SQL UPDATE statement rather than the INSERT statement. In this case, use SetItemStatus to change the row's status to DataModified!.

> **Tip**
> To reset the update status of the entire DataWindow object, use the ResetUpdate function. This sets all status flags to NotModified!.

Examples

This statement sets the status of row 5 in the Salary column of the primary buffer of dw_history to NotModified!:

```
dw_history.SetItemStatus(5, "Salary",  &
   Primary!, NotModified!)
```

This statement sets the status of row 5 in the emp_status column of the primary buffer of dw_new_hire to DataModified!:

```
dw_new_hire.SetItemStatus(5, "emp_status",  &
   Primary!, DataModified!)
```

This code sets the status of row 5 in the primary buffer of dw_rpt to DataModified! if its status is currently NewModified!:

```
dwItemStatus l_status

l_status = dw_rpt.GetItemStatus(5, 0, Primary!)

IF l_status = NewModified! THEN
 dw_rpt.SetItemStatus(5, 0, Primary!, DataModified!)
END IF
```

See also

GetItemStatus
ResetUpdate

SetLibraryList

Description

Changes the files in the library search path of the application.

Applies to

Application object

Syntax

applicationname.**SetLibraryList** (*filelist*)

Parameter	Description
applicationname	The name of the application object for which you want to change the library search path.
filelist	A comma-separated list of file names. Specify the full filename with its extension. If you do not specify a path, PowerBuilder uses the system's search path to find the file.

Return value

Integer. Returns *1* if it succeeds and *-1* if the application is being run from PowerBuilder, rather than from a standalone executable.

Usage

When your application needs to load an object, PowerBuilder searches for the object first in the executable file and then in the dynamic libraries specified for the application. You can change the order in which the libraries are searched with SetLibraryList. You can list the libraries in a new order or specify a different list of library files.

Calling SetLibraryList replaces the list of library files specified in the executable with a new list of files. PowerBuilder cannot check whether the libraries you specify are appropriate for the application. It is up to you to make sure the libraries contain the objects that the application needs.

The executable file is always first in the library search path. If you include it in *filelist*, it is ignored.

If you are running SetLibraryList from PowerBuilder, the function has no effect.

Example

This statement specifies three files in the library search path:

```
myapp.SetLibraryList("reports.pbd,dw.pbd,query.pbd")
```

SetMask

Description

Sets the edit mask and edit mask data type for an EditMask control.

Applies to

EditMask controls

Syntax

editmaskname.**SetMask** (*maskdatatype, mask*)

Parameter	Description
editmaskname	The name of the EditMask for which you want to specify the edit mask

Parameter	Description
maskdatatype	A MaskDataType enumerated data type indicating the data type of the mask. Values are: ♦ DateMask! ♦ DateTimeMask! ♦ DecimalMask! ♦ NumericMask! ♦ StringMask! ♦ TimeMask!
mask	A string whose value is the edit mask

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

In an edit mask, a fixed set of characters represent a type of character that the user can enter. In addition, punctuation controls the format of the entered value. Each mask data type has its own set of valid characters.

For example, the following is a mask of type string for a telephone number. The EditMask control displays the punctuation (the parentheses and dash). The pound signs represent the digits that the user will enter. The user cannot enter any characters other than digits:

```
(###) ###-####
```

For help in specifying a valid mask, see the Edit Mask Style dialog for an EditMask control in the Window painter. A listbox in the dialog shows the meaning of the special mask characters for each data type, as well as masks that have already been defined.

If you are specifying the mask for a number, the format must use U.S. notation. That is, comma represents the thousands delimiter and period represents the decimal place. During execution, the locally correct symbols will be displayed.

You cannot use color for edit masks as you can for display formats.

Examples

These statements set the mask for the EditMask password_mask to the mask in pword_code. The mask requires the user to enter a digit followed by 4 characters of any type:

```
string pword_code
pword_code = "#xxxx"
password_mask.SetMask(StringMask!, pword_code)
```

This statement sets the mask for the EditMask password_mask to a 5-digit numeric mask:

```
password_mask.SetMask(NumericMask!, "#####")
```

SetMicroHelp

Description

Specifies the text to be displayed in the MicroHelp box in an MDI frame window.

Applies to

MDI frame windows

Syntax

windowname.**SetMicroHelp** (*string*)

Parameter	Description
windowname	The name of the MDI frame window with MicroHelp for which you want to set the MicroHelp text
string	A string whose value is the new MicroHelp text

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The Tag attribute of a control is a useful place to store MicroHelp text. When the control gets the focus, you can use SetMicroHelp in the GetFocus event script to display the Tag attribute's text in the MicroHelp box on the window frame.

For menus, PowerBuilder automatically displays the MicroHelp text you have specified in the Menu painter when the user selects a MenuItem. You can use SetMicroHelp in the script for a MenuItem's Selected event to override the predefined MicroHelp and display some other text in the MicroHelp box. SetMicroHelp does not change the predefined MicroHelp text.

Examples

This statement changes the MicroHelp displayed in the frame of W_New to Delete selected text:

```
W_New.SetMicroHelp("Delete selected text")
```

In this example, the string Close the Window is a tag value associated with the CommandButton cb_done in W_New. In the script for the GetFocus event in cb_done, this statement displays Close the Window as MicroHelp in W_New when cb_done gets focus:

```
W_New.SetMicroHelp(This.Tag)
```

SetNull

Description

Sets a variable to NULL. The variable can be any data type.

Syntax

SetNull (*anyvariable*)

Parameter	Description
anyvariable	The variable you want to set to NULL

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Use SetNull to set a variable to NULL before writing it to the database. Note that PowerBuilder does not initialize variables to NULL; it initializes variables to the default initial value for the data type unless you specify a value when you declare the variable.

Example

This statement sets the variable Salary to NULL:

```
SetNull(Salary)
```

See also

IsNull

SetPicture

Description

Assigns a bitmap stored in a blob to be the image in a Picture control.

Applies to

Picture controls

Syntax

picturecontrol.**SetPicture** (*bitmap*)

Parameter	Description
picturecontrol	The name of a Picture control in which you want to set the bitmap.
bitmap	A blob containing the new bitmap. *Bitmap* must be a valid picture in bitmap (BMP), run-length encoded (RLE), Windows Metafile (WMF), or Macintosh PICT format.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If you use FileRead to get the bitmap image from a file, remember that the FileRead function can read a maximum of 32765 characters at a time. To check the length of a file, call FileLength. If the file is over 32765 characters, you can call FileRead more than once and concatenate the return values.

Example

These statements allow the user to select a file and then open the file and set the Picture control p_1 to the bitmap in the selected file:

```
integer fh, ret
blob Emp_pic
string txtname, named
string defext = "BMP"
string  Filter = "bitmap Files (*.bmp), *.bmp"

ret = GetFileOpenName("Open Bitmap", txtname,  &
   named, defext, filter)
IF ret = 1 THEN
   fh = FileOpen(txtname, StreamMode!)
   IF fh <> -1 THEN
      FileRead(fh, Emp_pic)
      FileClose(fh)
      p_1.SetPicture(Emp_pic)
   END IF
END IF
```

SetPointer

Description

Sets the mouse pointer to the specified shape.

Syntax

SetPointer (*type*)

Parameter	Description
type	A value of the Pointer enumerated data type indicating the type of pointer you want. Values are:  Arrow!  Cross!  Beam!  HourGlass!  SizeNS!  SizeNESW!  SizeWE!  SizeNWSE!  UpArrow!

Return value

Pointer. Returns the enumerated type of the pointer it replaced so the script can restore it, if necessary.

Usage

Use SetPointer to display an hourglass at the beginning of a script when the script will take a long time to execute.

The pointer remains set until you change it again in the script or the script terminates.

Note
The pointer automatically changes back to an arrow when the script finishes executing. You do not have to change it back to an arrow.

In PowerBuilder's painters, you can specify the pointer shape that PowerBuilder displays when the user moves the pointer over a window, a control, or specific parts of a DataWindow object. The available shapes include the stock pointers listed above, as well as any custom cursor files you have.

Examples

This statement sets the pointer to the hourglass shape:

```
SetPointer(HourGlass!)
```

This example saves the old pointer and restores it when a long activity is completed:

```
pointer oldpointer  // Declares a pointer variable
oldpointer = SetPointer(HourGlass!)
. . .  // Performs some long activity
SetPointer(oldpointer)
```

SetPosition

Description

For controls in a window, specifies the position of a control in the front-to-back order within a window. For a window, specifies whether it always displays on top of other open windows. For DataWindows, moves an object within the DataWindow to another band or changes the front-to-back order of objects within a band.

Applies to

(Syntax 1) A control within a window or a window

(Syntax 2) DataWindow controls

Syntax 1

objectname.**SetPosition** (*position* {, *precedingobject* })

Parameter	Description
objectname	The name of a control for which you want to specify a location in the front-to-back order within the window, or the name of a window for which you want to specify whether it will always display on top. *Objectname* cannot be a child window or a sheet.
position	A SetPosType enumerated data type. The values you can specify depend on whether *objectname* is a control or a window. For controls, values are: ♦ Behind! — Position *objectname* behind *precedingobject* in the order. ♦ ToTop! — Position *objectname* on top of all other controls. ♦ ToBottom! — Position *objectname* behind all other controls. For windows, values are: ♦ TopMost! — Always display *objectname* on top of all other open windows. ♦ NoTopMost! — Do not always display *objectname* on top of all other open windows.
precedingobject (optional)	The name of the object you want to position *objectname* behind. *Precedingobject* is required if *position* is Behind!.

Return value 1

Integer. Returns *1* when it succeeds and *-1* if an error occurs.

Syntax 2

datawindowname.**SetPosition** (*objectname, band, bringtofront*)

Parameter	Description
datawindowname	The name of the DataWindow control containing the object.
objectname	The name of the object within the DataWindow that you want to move. You assign names to the DataWindow objects in the DataWindow painter.

Parameter	Description
band	The name of the band or layer in which you want to position *objectname*: **Bands**: detail, header, footer, summary, header.#, trailer.# **Layers**: background, foreground # is the group level number. Enter the empty string ("") if you do not want to change the band.
bringtofront	A boolean indicating whether you want to bring *objectname* to the front within the band: TRUE — Bring it to the front. FALSE — Do not bring it to the front.

Return value 2

Integer. Returns *1* when it succeeds and *-1* if an error occurs.

Usage

The front-to-back order for controls determines which control covers another when they overlap. If a control completely covers another control, the control that is in back becomes inaccessible to the user.

When you specify TopMost! for more than one window, the most recently executed SetPosition function controls which window displays on top.

Examples

This statement positions cb_two on top:

```
cb_two.SetPosition(ToTop!)
```

This statement positions cb_two behind cb_three:

```
cb_two.SetPosition(Behind!, cb_three)
```

This statement makes the window w_signon the topmost window:

```
w_signon.SetPosition(TopMost!)
```

This statement makes the window w_signon no longer necessarily the topmost window:

```
w_signon.SetPosition(NoTopMost!)
```

Syntax 2

This statement moves oval_red in dw_rpt to the header and brings it to the front:

```
dw_rpt.SetPosition("oval_red", "header", TRUE)
```

This statement does not change the position of oval_red , but does bring it to the front:

```
dw_rpt.SetPosition("oval_red", "", TRUE)
```

This statement moves oval_red to the footer but does not bring it to the front:

```
dw_rpt.SetPosition("oval_red", "footer", FALSE)
```

SetProfileString

Description

Writes a value in a profile file for a PowerBuilder application.

Syntax

SetProfileString (*filename, section, key, value*)

Parameter	Description
filename	A string whose value is the name of the profile file. If you do not include the full path in *filename*, PowerBuilder searches the DOS path for *filename*.
section	A string whose value is the name of a group of related values in the profile file. If *section* does not exist in the file, PowerBuilder adds it.
key	A string whose value is the key in *section* for which you want to specify a value. If *key* does not exist in *section*, PowerBuilder adds it.
value	A string whose value is the value you want to specify for *key*.

Return value

Integer. Returns *1* when it succeeds and *-1* if it fails because *filename* is not found or cannot be accessed.

Usage

A profile file consists of section labels, which are enclosed in square brackets, and keys, which are followed by an equal sign and a value. By changing the values assigned to the keys, you can specify custom settings for each installation of your application. When you are planning your own profile file, you select the section and key names and determine how the values are used.

For example, a profile file might contain information about the user. In the sample below, User Info is the section name and the other values are the keys. There is no space before and after the equal sign:

```
[User Info]
Name="James Smith"
JobTitle="Window Washer"
SecurityClearance=9
Password=
```

Call SetProfileString to store configuration information, supplied by you or the user, in a profile file. You can call the functions ProfileInt and ProfileString to use that information to customize your PowerBuilder application during execution.

Example

This statement sets the keyword Title in section Position of file C:\PROFILE.INI to the string MGR:

```
SetProfileString("C:\PROFILE.INI",  &
   "Position", "Title", "MGR")
```

See also

ProfileInt
ProfileString

SetRedraw

Description

Controls the automatic redrawing of an object or control after each change to its attributes.

Applies to

Any object except a Menu

Syntax

objectname.**SetRedraw** (*boolean*)

Parameter	Description
objectname	The name of the object or control for which you want to change the redraw setting. *Objectname* can be a child DataWindow, but cannot be a menu.
boolean	A boolean value that controls whether PowerBuilder redraws an object automatically after a change. Values are: ♦ TRUE — Automatically redraw the object or control after each change to its attributes. ♦ FALSE — Do not redraw after each change.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

By default, PowerBuilder redraws a control after each change to attributes that affect appearance. Use SetRedraw to turn off redrawing temporarily in order to avoid flicker and reduce redrawing time when you are making several changes to the attributes of an object or control.

> ***Caution***
> *If you turn redraw off, you must turn it on again. Otherwise, problems may result. In addition, if redraw is off and you change the Visible or Enabled attribute of an object in the window, the tabbing order may be affected.*

Examples

This statement turns off redraw for lb_Location:

```
lb_Location.SetRedraw(FALSE)
```

This statement turns on redraw for lb_Location:

```
lb_Location.SetRedraw(TRUE)
```

If lb_Location is sorted (lb_Location.Sorted = TRUE), these statements use SetRedraw to avoid sorting and redrawing the list of lb_Location until all the new items have been added:

```
lb_Location.SetRedraw(FALSE)
lb_Location.AddItem("Atlanta")
lb_Location.AddItem("Boston")
lb_Location.AddItem("Washington")
lb_Location.SetRedraw(TRUE)
```

SetRemote

Description

Asks a DDE server application to accept data and store it in the specified location. There are two ways of calling SetRemote, depending on the type of DDE connection you've established:

- When you are making a single DDE request of a server application (a cold link), use Syntax 1.
- When you have established a warm link by opening a channel to the server application, use Syntax 2. A warm link, with an open channel, is more efficient when you intend to make several DDE requests.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax 1

SetRemote (*location, value, applname, topicname*)

Parameter	Description
location	A string whose value is the location of the data in the server application that will accept the data. The format of *location* depends on the application that will receive the request.
value	A string whose value you want to send to the remote application.
applname	A string whose value is the DDE name of the server application.
topicname	A string identifying the data or the instance of the application that will accept the data (for example, in Microsoft Excel, the topic name could be the name of an open spreadsheet).

Return value 1

Integer. Returns *1* if it succeeds and a negative integer if an error occurs. Values are:

- *-1* Link was not started
- *-2* Request denied

Syntax 2

SetRemote (*location*, *value*, *handle* {,*windowhandle*})

Parameter	Description
location	A string whose value is the location of the data in the server application that will accept the data. The format of *location* depends on the application that will receive the request.
value	A string whose value you want to send to the remote application.
handle	A long that identifies the channel to the DDE server application. *Handle* is the value returned by OpenChannel, which you call to open a DDE channel.
windowhandle (optional)	The handle to the window that is acting as the DDE client.

Return value 2

Integer. Returns *1* if it succeeds and a negative integer if an error occurs. Values are:

- *-1* Link was not started
- *-2* Request denied
- *-9* handle is NULL

Usage

When using DDE, your PowerBuilder application must have an open window, which will be the client window. For Syntax 1, the active window is the DDE client window. For Syntax 2, you can specify a different client window with the *windowhandle* argument.

Before using Syntax 2 of SetRemote, call OpenChannel to establish a DDE channel.

For more information about DDE channels and warm and cold links, see the ExecRemote function.

Example

This statement asks Microsoft Excel to set the value of the data in row 5, column 7 of a worksheet called SALES.XLS to 4500:

```
SetRemote("R5C7", "4500", "Excel", "SALES.XLS")
```

Syntax 2

This example opens a channel to a Microsoft Excel worksheet and asks it to set the value of the data in row 5 column 7 to 4500:

```
long handle
handle = OpenChannel("Excel", "REGION.XLS")
SetRemote("R5C7", "4500", handle)
```

See also

ExecRemote
GetRemote
OpenChannel

SetRow

Description

Sets the current row in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetRow** (*row*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the current row
row	A long whose value is the row you want to make current

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. If *row* is less than 1 or greater than the number of rows, SetRow fails.

Usage

SetRow moves the cursor to the current row but does not scroll the DataWindow control.

Events

SetRow may trigger these events:

- ItemChanged
- ItemError
- ItemFocusChanged
- RowFocusChanged

Avoiding infinite loops
Never call SetRow in the ItemChanged event or any of the other events listed above. Because SetRow can trigger these events, such a recursive call can cause a stack fault.

Examples

This statement sets the current row in dw_employee to 15:

```
dw_employee.SetRow(15)
```

This example unhighlights all rows, if any. Then it sets the current row to 15 and highlights it. If row 15 is not visible, you can use ScrollToRow instead of SetRow:

```
dw_employee.SelectRow(0, FALSE)
dw_employee.SetRow(15)
dw_employee.SelectRow(15, TRUE)
```

See also

GetColumn
GetRow
SetColumn
SetRowFocusIndicator

SetRowFocusIndicator

Description

Specifies the visual indicator that identifies the current row in the DataWindow control. You can use Windows' standard dotted-line rectangle, PowerBuilder's pointing hand, or an image stored in a Picture control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetRowFocusIndicator** (*focusindicator* &
{, *xlocation* {, *ylocation* }})

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the row focus indicator.
focusindicator	The visual indicator for the current row. Valid values are a RowFocusInd enumerated data type or the name of a Picture control, as follows: ♦ Off! — No indicator. ♦ FocusRect! — Put a dotted line rectangle around the row. FocusRect! has no effect on the Macintosh.

Parameter	Description
	♦ Hand! — Use the PowerBuilder pointing hand. ♦ Name of a Picture control — Use the specified Picture control.
xlocation (optional)	An integer whose value is the x coordinate in PowerBuilder units of the position of the hand or bitmap relative to the upper-left corner of the row.
ylocation (optional)	An integer whose value is the y coordinate in PowerBuilder units of the position of the hand or bitmap relative to the upper-left corner of the row.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Sets the current row indicator in *datawindowname* to *focusindicator*. If you select Hand or a Picture control as the indicator, PowerBuilder displays the indicator at the left side of the body of the DataWindow unless you specify location coordinates (*xlocation*, *ylocation*). The default location is 0,0 (the left side of the body of the DataWindow control).

> **Pictures as row focus indicators**
> To use a picture as the row focus indicator, set up the Picture control in the Window painter. Place the Picture control in the window that contains the DataWindow control and then reference it in the SetRowFocusIndicator function. You can hide the picture or place it under the DataWindow control so the user doesn't see the control itself.

Examples

This statement sets the row focus indicator in dw_employee to the pointing hand:

```
dw_employee.SetRowFocusIndicator(Hand!)
```

If p_arrow is a Picture control in the window, the following statement sets the row focus indicator in dw_employee to p_arrow:

```
dw_employee.SetRowFocusIndicator(p_arrow)
```

See also

GetRow
SetRow

SetSeriesStyle

Description

Specifies the appearance of a series in a graph. There are several syntaxes, depending on what settings you want to change:

- To set the series' colors, use Syntax 1.
- To set the line style and width, use Syntax 2.
- To set the fill pattern or symbol for the series, use Syntax 3.
- To specify whether the series is an overlay (that is, whether it is shown in front of other data as a line), use Syntax 4.

Applies to

Graph controls in windows and user objects, and graphs in DataWindow controls

Syntax 1

controlname.**SetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *colortype*, *color*)

Parameter	Description
controlname	The name of the graph in which you want to set the color of a series, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control for which you want to set the color of a series.
seriesname	A string whose value is the name of the series for which you want to set the color.
colortype	A value of the grColorType enumerated data type specifying the item for which you want to set the color. Values are: ♦ Foreground! — Text color ♦ Background! — Background color ♦ LineColor! — Line color ♦ Shade! — Shade (for graphics that are three-dimensional or have solid objects)
color	A long specifying the new color for *colortype*

Return value 1

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 2

controlname.**SetSeriesStyle** ({*graphcontrol*, } *seriesname*, & *linestyle*, *linewidth*)

Parameter	Description
controlname	The name of the graph in which you want to set the line style and width of a series, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the line style and width.
seriesname	A string whose value is the name of the series for which you want to set the line style and width.
linestyle	A value of the LineStyle enumerated data type. Values are: ♦ Continuous! ♦ Dash! ♦ DashDot! ♦ DashDotDot! ♦ Dot! ♦ Transparent!
linewidth	An integer specifying the width of the line in pixels.

Return value 2

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 3

controlname.**SetSeriesStyle** ({*graphcontrol*, } *seriesname*, & *enumvalue*)

Parameter	Description
controlname	The name of the graph in which you want to set the appearance of a series, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the appearance.
seriesname	A string whose value is the name of the series in which you want to set the appearance.
enumvalue	A value of an enumerated data type specifying an appearance setting for the series. Values for the FillPattern or grSymbolType enumerated data types follow.

Parameter	Description
	To change the fill pattern, use a FillPattern value: ♦ Bdiagonal! (Lines from lower left to upper right) ♦ Diamond! ♦ Fdiagonal! (Lines from upper left to lower right) ♦ Horizontal! ♦ Solid! ♦ Square! ♦ Vertical! To change the symbol type, use a grSymbolType value: ♦ NoSymbol! ♦ SymbolHollowBox! ♦ SymbolX! ♦ SymbolStar! ♦ SymbolHollowUpArrow! ♦ SymbolHollowCircle! ♦ SymbolHollowDiamond! ♦ SymbolSolidDownArrow! ♦ SymbolSolidUpArrow! ♦ SymbolSolidCircle! ♦ SymbolSolidDiamond! ♦ SymbolPlus! ♦ SymbolHollowDownArrow! ♦ SymbolSolidBox!

Return value 3

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Syntax 4

controlname.**SetSeriesStyle** ({ *graphcontrol*, } *seriesname*, & *overlaystyle*)

Parameter	Description
controlname	The name of the graph in which you want to set the overlay status of a series, or the name of the DataWindow control containing the graph.
graphcontrol (DataWindow control only)	A string whose value is the name of the graph in the DataWindow control in which you want to set the overlay status.

Parameter	Description
seriesname	A string whose value is the name of the series whose overlay status you want to change.
overlaystyle	A boolean value indicating whether you want the series to be an overlay, meaning that the series is shown in front as a line. Set *overlaystyle* to TRUE to make the specified series an overlay. Set it to FALSE to remove the overlay setting.

Return value 4

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

For a graph in a DataWindow, you can specify the appearance of a series in the graph before PowerBuilder draws the graph. To do so, define a user event for pbm_dwngraphcreate and call SetSeriesStyle in the script for that event. The event pbm_dwngraphcreate is triggered just before a graph is created in a DataWindow object.

Examples

This statement sets the text (foreground) color of the series named Salary in the graph gr_emp_data to black:

```
gr_emp_data.SetSeriesStyle("Salary",  &
    Foreground!, 0)
```

This statement sets the background color of the series named Salary in the graph gr_depts in the DataWindow control dw_employees to black:

```
dw_employees.SetSeriesStyle("gr_depts",  &
   "Salary", Background!, 0)
```

These statements in the Clicked event of the graph control gr_product_data coordinate line color between it and the graph gr_sales_data. The script stores the line color for the series under the mouse pointer in the graph gr_product_data in the variable line_color. Then it sets the line color for the series northeast in the graph gr_sales_data to that color:

```
string  SeriesName
integer SeriesNbr, Series_Point
long line_color
grObjectType MouseHit

MouseHit = ObjectAtPointer(SeriesNbr,Series_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName =  &
      gr_product_data.SeriesName(SeriesNbr)
```

```
   gr_product_data.GetSeriesStyle(SeriesName,  &
      LineColor!, line_color)

   gr_sales_data.SetSeriesStyle("Northeast", &
      LineColor!, line_color)
END IF
```

Syntax 2

This statement sets the line style and width for the series named Costs in the graph gr_product_data:

```
gr_product_data.SetSeriesStyle("Costs",  &
   Dot!, 5)
```

Syntax 3

This statement sets the symbol used for the series named Costs in the graph gr_product_data to a plus sign:

```
gr_product_data.SetSeriesStyle("Costs",  &
   SymbolPlus!)
```

This statement sets the symbol used for the series named Costs in the graph gr_computers in the DataWindow control dw_equipment to X:

```
dw_equipment.SetSeriesStyle("gr_computers",  &
   "Costs", SymbolX!)
```

Syntax 4

This statement sets the style of the series named Costs in the graph gr_product_data to overlay:

```
gr_product_data.SetSeriesStyle("Costs", TRUE)
```

These statements in the Clicked event of the DataWindow control dw_employees store the style of the series under the pointer in the graph gr_depts in the variable style_type. If the style of the series is overlay (TRUE), the script changes the style to normal (FALSE):

```
string  SeriesName
integer SeriesNbr, Data_Point
boolean overlay_style
grObjectType MouseHit

MouseHit = dw_employees.ObjectAtPointer(  &
   "gr_depts", SeriesNbr, Data_Point)

IF MouseHit = TypeSeries! THEN
   SeriesName = &
      dw_employees.SeriesName("gr_depts",SeriesNbr)

   dw_employees.GetSeriesStyle("gr_depts",  &
      SeriesName, overlay_style)

   IF overlay_style then  &
      dw_employees.SetSeriesStyle("gr_depts",  &
      SeriesName, FALSE)
END IF
```

See also

GetDataStyle
SeriesName
GetSeriesStyle
SetDataStyle

SetSort

Description

Specifies sort criteria for a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetSort** (*format*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to define the sort criteria.
format	A string whose value is valid sort criteria for the DataWindow (see Usage). The expression includes column names or numbers. A column number must be preceded by a pound sign (#). If *format* is NULL, PowerBuilder prompts you to enter the sort criteria.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A DataWindow object can have sort criteria specified as part of its definition. SetSort overrides the definition, providing new sort criteria for the DataWindow. However, it does not actually sort the rows. Call the Sort function to perform the actual sorting.

The sort criteria for a column has one of the forms shown in the following table, depending on whether you specify the column by name or number. *Order* is either A for ascending or D for descending order. You can specify secondary sorting by specifying criteria for additional columns in the format string. Separate each column specification with a comma.

Syntax for sort order	Examples
columnname order	"emp_lname A" "emp_lname A, dept_id D"
#columnnumber order	"#3 A"

Letting the user specify sort criteria
If you specify a null value for *format*, PowerBuilder displays the Specify Sort Columns dialog when you call Sort, allowing users to specify their own sort criteria (see Sort).

Examples

This statement sets the sort criteria for dw_employee so emp_status is sorted in ascending order and within each employee status, emp_salary is sorted in descending order:

```
dw_employee.SetSort("emp_status A, emp_salary D")
```

If emp_status is column 1 and emp_salary is column 5 in dw_employee, then the following statement is equivalent to the sort specification above:

```
dw_employee.SetSort("#1 A, #5 D")
```

This example defines sort criteria to sort the status column in ascending order and the salary column in descending order within status. After assigning the sort criteria to the DataWindow control dw_emp, it sorts dw_emp:

```
string newsort
newsort = "emp_status A, emp_salary D"
dw_emp.SetSort(newsort)
dw_emp.Sort( )
```

See also

Sort

SetSQLPreview

Description

Specifies the SQL statement for a DataWindow control that PowerBuilder is about to send to the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetSQLPreview** (*sqlsyntax*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to set the SQL statement that will be submitted to the database server
sqlsyntax	A string whose value is valid SQL syntax for the SQL statement that will be submitted to the database server

Usage

Use SetSQLPreview to modify syntax before you update the database with changes in the DataWindow object.

To obtain the current SQL statement, call GetSQLPreview before calling this function.

> **Note**
> Call this function *only* in the script for the SQLPreview event.

Return value

Integer. Returns *1* if it succeeds and *0* if an error occurs.

Example

This statement sets the current SQL string for the DataWindow dw_1:

```
dw_1.SetSQLPreview(  &
   "INSERT INTO billings VALUES(100, " +  &
   String(Current_balance) + ")")
```

See also

GetSQLPreview
GetUpdateStatus

SetSQLSelect

Description

Specifies the SQL SELECT statement for a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetSQLSelect** (*statement*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to change the SELECT statement.
statement	A string whose value is the SELECT statement for the DataWindow object. The statement must structurally match the current SELECT statement (that is, it must return the same number of columns, the columns must be the same data type, and the columns must be in the same order).

Return value

Integer. SetSQLSelect returns *1* if it succeeds and *-1* if the SELECT statement cannot be changed.

Usage

Use SetSQLSelect to dynamically change the SQL SELECT statement for a DataWindow object in a script.

If the DataWindow is updatable, PowerBuilder validates the SELECT statement against the database and DataWindow column specifications when you call the SetSQLSelect function. Each column in the SQL SELECT statement must match the column type in the DataWindow object. The statement is validated *only* if the DataWindow object is updatable.

You must use the SetTrans or SetTransObject function to set the DataWindow control's internal transaction object before the SetSQLSelect function will execute.

If the new SELECT statement has a different table name in the FROM clause and the DataWindow object is updatable, then PowerBuilder must change the update information for the DataWindow object. PowerBuilder assumes the key columns are in the same positions as in the original definition. The following conditions will make the DataWindow not updatable:

- There is more than one table in the FROM clause.
- A DataWindow update column is a computed column in the SELECT statement.

If changing the SELECT statement makes the DataWindow object not updatable, the DataWindow control cannot execute an Update function call for the DataWindow object in the future.

Limitations to using SetSQLSelect
Use SetSQLSelect *only* if the data source for the DataWindow object is a SQL SELECT statement *without* arguments and you want PowerBuilder to modify the update information for the DataWindow object. are changing the UPDATE statement. If the SELECT statement has arguments, use Modify. For example:

```
dw_1.Modify("DataWindow.Table.Select='select...'")
```

However, Modify will not verify SELECT statement or change the update information, making it faster but more susceptible to user error.

Example

If the current SELECT statement for dw_emp retrieves no rows, the following statements replace it with the syntax in NewSyn:

```
string OldSyn, NewSyn

OldSyn = &
'SELECT employee.EMP_Name FROM employee'  &
   + 'WHERE salary < 70000'
NewSyn = 'SELECT employee.EMP_Name FROM employee'  &
   + 'WHERE salary < 100000'

IF dw_emp.Retrieve( ) = 0 THEN
   dw_emp.SetSQLSelect(NewSyn)
   dw_emp.Retrieve()
END IF
```

See also

Modify
Retrieve
SetTrans
SetTransObject
Update

SetState

Description

Sets the highlighted state of an item in a ListBox. SetState is only applicable to a ListBox control whose MultiSelect attribute is set to TRUE.

Applies to

ListBox controls

Syntax

listboxname.**SetState** (*index*, *state*)

Parameter	Description
listboxname	The name of the ListBox in which you want to set the state (highlighted or not highlighted) for an item. The ListBox's MultiSelect attribute must be set to TRUE.
index	The number of the item for which you want to set the state. Specify 0 to set the state of all the items in the ListBox.
state	A boolean value that determines the state of the item: ♦ TRUE — Selected ♦ FALSE — Not selected

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

When a ListBox's MultiSelect attribute is FALSE, use SelectItem, instead of SetState, to select one item at a time.

Examples

This statement turns on the highlight for item 6 in lb_Actions:

```
lb_Actions.SetState(6, TRUE)
```

This statement deselects all items in lb_Actions:

```
lb_Actions.SetState(0, FALSE)
```

This statement turns off the highlight for item 6 in lb_Actions if it is selected and turns it on again if it is not selected:

```
lb_Actions.State(6) = 1 THEN
   lb_Actions.SetState(6, FALSE)
ELSE
   lb_Actions.SetState(6, TRUE)
END IF
```

See also

SelectItem
SetTop
State

SetTabOrder

Description

Changes the tab sequence number of a column in a DataWindow control to the specified value.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetTabOrder** (*column, tabnumber*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to define the tab order.
column	The column to which you are assigning a tab value. *Column* can be a column number (integer) or a column name (string).
tabnumber	The tab sequence number (0 - 9999) you want to assign to the DataWindow column. 0 removes the column from the tab order, which makes it read-only.

Return value

Integer. Returns the previous tab value of the column if it succeeds and *-1* if an error occurs.

Usage

Use SetTabOrder to change a column in a DataWindow object to read-only by changing the tab sequence number of the column to 0.

Examples

This statement changes column 4 of dw_Employee to read-only:

```
dw_Employee.SetTabOrder(4, 0)
```

These statements change column 4 of dw_employee to read-only and later restore the column to its original tab value with read/write status:

```
integer OldTabNum
// Set OldTabNum to the previous tab order value
OldTabNum = dw_employee.SetTabOrder(4, 0)
. . .  // Some processing
// Return column 4 to its previous tab value.
dw_employee.SetTabOrder(4, OldTabNum)
```

SetText

Description

Replaces the text in the edit control over the current row and column in a DataWindow control.

Applies to

DataWindow controls

Syntax

datawindowname.**SetText** (*text*)

Parameter	Description
datawindowname	The name of the DataWindow control in which you want to specify the text in the current row and column.
text	A string whose value you want to put in the current row and column. The value must be compatible with the data type of the column.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

SetText only sets the value in the DataWindow's edit control. When the user changes focus to another row and column, PowerBuilder accepts the text as the item in the row and column.

In the ItemChanged or ItemError event, PowerBuilder or your own script may determine that the value in the edit control is invalid or that it needs further processing. You can call SetItem to specify a new item value for the row and column. After calling SetItem, you can call SetText to put that same value in the edit control so that the user sees the value too. In the script, set the action code to reject the value in the edit control, avoiding further processing, and allow the focus to change. (Call SetActionCode with a value of 2 for ItemChanged and a value of 3 for ItemError.)

Examples

These statements replace the value of the current row and column in dw_employee with Tex and then call AcceptText to accept and move Tex into the current column. (Do not use this code in the ItemChanged or ItemError event because it calls AcceptText):

```
dw_employee.SetText("Tex")
dw_employee.AcceptText()
```

This example converts a number that the user enters in the column called credit to a negative value and sets both the item and the edit control's text to the negative number. This code is the script for the ItemChanged event.

```
integer negative

IF This.GetColumn() = "credit" THEN
   IF Integer(This.GetText()) > 0 THEN
      // Convert to negative if it's positive
      negative = Integer(This.GetText()) * -1
   END IF

   //  Change the primary buffer value.
   This.SetItem(This.GetRow(),"credit", negative)

   // Change the value in the edit control
   This.SetText(String(negative))
   This.SetActionCode(2)
END IF
```

See also

AcceptText
GetText
SetActionCode

SetTop

Description

Scrolls a ListBox control so that the specified item is the first visible item.

Applies to

ListBox controls

Syntax

listboxname.**SetTop** (*index*)

Parameter	Description
listboxname	The name of the ListBox that you want to scroll
index	The number of the item you want to become the first visible item

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Examples

This statement scrolls item 6 in lb_Actions to the top of the ListBox so that it is the first visible item:

```
lb_Actions.SetTop(6)
```

The following statement scrolls the currently selected item in lb_Actions to the top of the list of items:

```
lb_Actions.SetTop(lb_Actions.SelectedIndex())
```

See also

SetFocus
SetState

SetTrans

Description

Sets the values in the internal transaction object for a DataWindow control to the values from in the specified transaction object. The transaction object supplies connection settings, such as the database name.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetTrans** (*transaction*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the values of the internal transaction object
transaction	The name of the transaction object from which you want *datawindowname* to get values

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

In most cases, use the SetTransObject function to specify the transaction object. It is more efficient and allows you to control when changes to the database get committed.

SetTrans copies the values from a specified transaction object to the DataWindow control's internal transaction object. When you use SetTrans in a script, the DataWindow uses its internal transaction object and automatically connects and disconnects as needed; any errors that occur cause an automatic rollback. With SetTrans, you do not specify SQL statements, such as CONNECT, COMMIT, and DISCONNECT. The DataWindow control connects and disconnects after each Retrieve or Update function.

Use SetTrans when you want PowerBuilder to manage the database connections automatically because you have a limited number of available connections or will be use the application from a remote location. SetTrans is appropriate when you are only retrieving data and do not need to hold database locks on records the user is modifying. However, for better performance, you should use SetTransObject.

Tips

You must set the parameters required to connect to your DBMS in the transaction object before you can use the transaction object to set the DataWindow's internal transaction object and connect to the database.

When you use SetTrans to specify the transaction object, you cannot update multiple DataWindow objects or multiple tables within one object.

Examples

This statement sets the values in the internal transaction object for dw_employee to the values in the default transaction object SQLCA:

```
dw_employee.SetTrans(SQLCA)
```

The following statements change the database type and password of dw_employee. The first two statements create the transaction object emp_TransObj. The next statement uses the GetTrans function to store the values of the internal transaction object for dw_employee in emp_TransObj. The next two statements change the database type and password. The SetTrans function assigns the revised values to dw_employee:

```
// Name the transaction object.
transaction emp_TransObj

// Create the transaction object.
emp_TransObj = CREATE transaction

// Fill the new object with the original values.
dw_employee.GetTrans(emp_TransObj)
// Change the database type.
emp_TransObj.DBMS ="Sybase"
```

```
// Change the password.
emp_TransObj.LogPass = "cam2"

// Put the revised values into the
// DataWindow transaction object.
dw_employee.SetTrans(emp_TransObj)
```

See also

GetTrans
SetTransObject

SetTransObject

Description

Causes a DataWindow control to use a programmer-specified transaction object. The transaction object provides the information necessary for communicating with the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetTransObject** (*transaction*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to use a programmer-specified transaction object rather than the DataWindow control's internal transaction object
transaction	The name of the transaction object you want to use in the *datawindowname*

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A programmer-specified transaction object gives you more control over the database transactions and provides efficient application performance. You control the database connection by using SQL statements such as CONNECT, COMMIT, and ROLLBACK.

Since the DataWindow control does not have to connect to the database for every RETRIEVE and UPDATE statement, these statements run faster. You are responsible for committing and rolling back transactions after you call the Update function, using code like the following:

```
IF dw_Employee.Update()>0 THEN
   COMMIT USING emp_transobject;
ELSE
   ROLLBACK USING emp_transobject;
END IF
```

You must set the parameters required to connect to your DBMS in the transaction object before you can use the transaction object to connect to the database. PowerBuilder provides a global transaction object called SQLCA, which is all you need if you are connecting to one database. You can also create additional transaction objects, as shown in the examples.

To use SetTransObject, write code that does the following tasks:

1 Set up the transaction object by assigning values to its fields (usually in the application's Open event).

2 Connect to the database using the SQL CONNECT statement and the transaction object (in the Open event for the application or window).

3 Call SetTransObject to associate the transaction object with the DataWindow control (usually in the window's Open event).

4 Check the return value from the Update function and follow it with an SQL COMMIT or ROLLBACK statement, as appropriate.

If you change the DataWindow object associated with the DataWindow control or if you disconnect and reconnect to a database, the connection between the DataWindow control and the transaction object is severed. You must call SetTransObject again to reestablish the connect.

SetTransObject versus SetTrans

In most cases, use the SetTransObject function to specify the transaction object because it is efficient and gives you control over when transactions are committed.

The SetTrans function provides another way of managing the database connection. SetTrans, which sets transaction information in the DataWindow control's internal transaction object, manages the connection automatically. You do not explicitly connect to the database; the DataWindow connects and disconnects for each database transaction, which is less efficient but necessary in some situations. See SetTrans for more information.

Examples

This statement causes dw_employee to use the default transaction object SQLCA:

```
dw_employee.SetTransObject(SQLCA)
```

This statement causes dw_employee to use the programmer-defined transaction object emp_TransObj. In this example, emp_TransObj is an instance variable, but your script must allocate memory for it with the CREATE statement before you use it:

```
emp_TransObj = CREATE transaction
. . .  // Assign values to the transaction object
dw_employee.SetTransObject(emp_TransObj)
```

This example has two parts. The first script, for the application's Open event, reads database parameters from an initialization file called MYAPP.INI and stores the values in the default transaction object (SQLCA). The Database section of MYAPP.INI has the same keywords as PowerBuilder's own PB.INI file. The parameters shown are for a SQL Server, ORACLE, or SQLBase database. The second script, for the window's Open event, establishes a connection and retrieves data from the database.

The application's Open event script populates SQLCA:

```
SQLCA.DBMS = ProfileString("myapp.ini",  &
      "database", "DBMS", " ")
SQLCA.Database = ProfileString("myapp.ini",  &
      "database", "Database", " ")
SQLCA.LogId = ProfileString("myapp.ini",  &
      "database", "LogId", " ")
SQLCA.LogPass = ProfileString("myapp.ini",  &
      "database", "LogPassword", " ")
SQLCA.ServerName = ProfileString("myapp.ini",  &
      "database", "ServerName", " ")
SQLCA.UserId = ProfileString("myapp.ini",  &
      "database", "UserId", " ")
SQLCA.DBPass = ProfileString("myapp.ini",  &
      "database", "DatabasePassword", " ")
SQLCA.lock = ProfileString("myapp.ini",  &
      "database", "lock", " ")
```

The Open event script for the window that contains the DataWindow control connects to the database, assigns the transaction object to the DataWindow, and retrieves data:

```
long RowsRetrieved
string LastName

// Connect to the database.
CONNECT USING SQLCA;
```

```
// Test whether the connect succeeded.
IF SQLCA.SQLCode <> 0 THEN
   MessageBox("Connect Failed", &
      "Cannot connect to database " &
      + SQLCA.SQLErrText)
   RETURN
END IF

// Set the transaction object to SQLCA.
dw_employee.SetTransObject(SQLCA)

// Retrieve the rows.
LastName = . . .
RowsRetrieved = dw_employee.Retrieve(LastName)
// Test whether the retrieve succeeded.
IF RowsRetrieved < 0 THEN
   MessageBox("Retrieve Failed",  &
      "Cannot retrieve data from the database.")
END IF
```

See also

GetTrans
SetTrans

SetValidate

Description

Sets the input validation rule for a column in a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**SetValidate** (*column*, *rule*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the input validation rule for a column.
column	The column for which you want to set the input validation rule. *Column* can be a column number (integer) or a column name (string).
rule	A string whose value is the validation rule for validating the data.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

Validation rules are boolean expressions that usually compare the value in the column's edit control to some other value. When data the user enters fails to meet the criteria established in the validation rule, an ItemError event occurs.

You can specify validation rules in the Database painter or the DataWindow painter, and you can change the rules in scripts using SetValidate. A validation rule can include any DataWindow painter function. See Chapter 2, "DataWindow Painter Functions," and the *User's Guide* for more information.

If you want to change a column's validation rule temporarily, you can use GetValidate to get and save the current rule.

To include the value the user entered in the validation rule, use the GetText function. You can compare its return value to the validation criteria.

If the validation rule contains numbers, the DataWindow expects the numbers in U.S. format. Be aware that the String function formats numbers using the current system settings. If you use it to build the rule, specify a display format that produces U.S. notation.

Examples

The following assigns a validation rule to the current column in dw_employee. The rule ensures that the data entered is greater than 0:

```
dw_employee.SetValidate(dw_employee.GetColumn(),  &
   "Number(GetText( )) > 0")
```

The following assigns a validation rule to the current column in dw_employee. The rule checks that the value entered is less than the value in the Full_Price column:

```
dw_employee.SetValidate(dw_employee.GetColumn(),  &
   "Number(GetText( )) < Full_Price")
```

This example defines a new validation rule for the column emp_state in the DataWindow control dw_employee. The new rule is [A-Z]+ , meaning the data in emp_state must be all uppercase characters. The text pattern must be enclosed in quotes within the quoted validation rule. The embedded quotes are specified with ~". The script saves the old rule, assigns the new rule, performs some processing, and then sets the validation rule back to the old rule:

```
string  OldRule, NewRule
NewRule = "Match(GetText ( ), ~"[A-Z]+~")"
OldRule = dw_employee.GetValidate("emp_state")
dw_employee.SetValidate("emp_state", NewRule)
. . .  //Process data using the new rule.
// Set the validation rule back to the old rule.
dw_employee.SetValidate("emp_state", OldRule)
```

See also GetValidate

SetValue

Description Sets the value of an item in a code table or the listbox portion of a DropDownListBox for a column in a DataWindow control.

Applies to DataWindow controls and child DataWindows

Syntax *datawindowname*.**SetValue** (*column*, *index*, *value*)

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to set the value of an item in a value list or code table.
column	A string whose value is the column that contains the value list or code table. *Column* can be a column number (integer) or a column name (string).
index	The number of the item in the value list or code table for which you want to set the value.
value	A string whose value is the new value for the item. For a code table, use a tab (~t) to separate the display value from the data value ("Texas~tTX"). The data value must be a string that can be converted to the data type of the column.

Return value Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Examples

This statement sets the value of item 3 in the value list for the column emp_state of dw_employee to Texas:

```
dw_employee.SetValue("emp_state", 3, "Texas")
```

This statement sets the display value of item 3 in the code table for the column named emp_state of dw_employee to Texas and the data value to TX:

```
dw_employee.SetValue("emp_state", 3, "Texas~tTX")
```

The following statements use an SQL cursor and FETCH statement to populate the ListBox portion of a DropDownListBox style column called product_col of a DataWindow object with code table values:

```
integer prod_code, i = 1
string prod_name

DECLARE prodcur CURSOR FOR
   SELECT product.name, product.code
   FROM product USING SQLCA;

CONNECT USING SQLCA;
IF SQLCA.SQLCode <> 0 THEN
   MessageBox("Status","Connect Failed "  &
      + SQLCA.SQLErrText)
   RETURN
END IF

OPEN prodcur;
IF SQLCA.SQLCode <> 0 THEN
   MessageBox("Status","Cursor Open Failed "  &
      + SQLCA.SQLErrText)
   RETURN
END IF

FETCH prodcur INTO :prod_name, :prod_code;

DO WHILE SQLCA.SQLCode = 0
   dw_products.SetValue("product_col", i,  &
      prod_name + "~t" + String(prod_code))
   i = i + 1
   FETCH prodcur INTO :prod_name, :prod_code;
LOOP

CLOSE prodcur;
DISCONNECT USING SQLCA;
```

See also

GetValue

ShareData

Description

Shares data retrieved by one DataWindow control, the primary DataWindow, with another DataWindow control, the secondary DataWindow. The controls do not share formatting; only the data is shared, including data in the primary buffer, the delete buffer, the filter buffer, and the sort order.

Applies to

DataWindow controls and child DataWindows

Syntax

dwprimary.**ShareData** (*dwsecondary*)

Parameter	Description
dwprimary	The name of the primary DataWindow control; the owner of the data. When you destroy this DataWindow, the data disappears. *Dwprimary* can be a child DataWindow.
dwsecondary	The name of the secondary DataWindow control; the control *dwprimary* will share the data with. The secondary DataWindow control cannot be a Crosstab DataWindow. It can be a child DataWindow.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

The columns must be the same for the DataWindow objects in the primary and secondary DataWindow controls, but the SELECT statements may be different. For example, you could share data between DataWindow objects with these SELECT statements:

```
SELECT dept_id from dept
SELECT dept_id from dept where dept_id = 200
SELECT dept_id from employee
```

You could also share data with a DataWindow object that has a script data source and a column defined to be like the dept_id column.

To share data between a primary DataWindow and more than one secondary DataWindow control, call ShareData for each secondary DataWindow control.

> **Note**
> You cannot use ShareData with Crosstab DataWindows.

To turn off sharing in a primary or secondary DataWindow control, call the ShareDataOff function. When sharing is turned off for the primary DataWindow control, the secondary DataWindows are disconnected and the data disappears. However, turning off sharing for a secondary DataWindow control does not affect the data in the primary DataWindow or other secondary DataWindows.

When you call functions in either the primary or secondary DataWindow control that change the data, PowerBuilder applies them to the primary DataWindow control and all secondary DataWindow controls are affected. For example, when you call any of the following functions for a secondary DataWindow control, PowerBuilder applies it to the primary DataWindow. Therefore, all messages normally associated with the function go to the primary DataWindow control:

Filter	Retrieve
ImportFile	SetFilter
ImportString	SetSort
ImportClipboard	Sort

Examples

In this example, the programmer wants to allow the user to view two portions of the same data retrieved from the database and uses the ShareData function to accomplish this in the script for the Open event for the window. The SELECT statement for both DataWindow objects is the same, but the DataWindow object in dw_dept displays only two of the five columns displayed in dw_employee.

```
CONNECT USING SQLCA;
dw_employee.SetTransObject(SQLCA)
dw_employee.Retrieve()
dw_employee.ShareData(dw_dept)
```

These statements share data between two DataWindow controls in different sheets within an MDI frame window:

```
CONNECT USING SQLCA;
mdi_sheet_1.dw_dept.SetTransObject(SQLCA)
mdi_sheet_1.dw_dept.Retrieve()
mdi_sheet_1.dw_dept.ShareData(mdi_sheet_2.dw_dept)
```

See also

ShareDataOff

ShareDataOff

Description

Turns off the sharing of data buffers for a DataWindow control.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**ShareDataOff ( )**

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow for which you want to turn off data sharing

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

One or more DataWindow controls can share data with a DataWindow control, which is called the primary DataWindow. See ShareData for more information about shared data buffers and primary and secondary DataWindows.

When you call ShareDataOff for a secondary DataWindow, that control no longer contains data, but the primary DataWindow and other secondary controls are not affected. When you call ShareDataOff for the primary DataWindow, all secondary DataWindow controls are disconnected and no longer contain data.

Example

These statements establish the sharing of data among three DataWindow controls and then turn off sharing for one of the secondary DataWindow controls:

```
CONNECT USING SQLCA;
dw_corp.SetTransObject(SQLCA)
dw_corp.Retrieve()
dw_corp.ShareData(dw_emp)
dw_corp.ShareData(dw_dept)
...  // Some processing
dw_emp.ShareDataOff( )
```

See also

ShareData

Show

Description

Makes an object or control visible, if it is hidden. If the object is already visible, Show brings it to the top.

Applies to

Any object

Syntax

objectname.**Show** ( )

Parameter	Description
objectname	The name of the object or control you want to make visible (show)

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If the specified object is a window that is not open, an execution error occurs.

You cannot use Show to show a dropdown or cascading menu, or any menu that has an MDI frame window as its parent window.

Equivalent syntax

You can set the object's Visible attribute instead of calling Show:

objectname.Visible = TRUE

This statement:

```
m_status.m_options.Visible = TRUE
```

is equivalent to:

```
m_status.m_options.Show( )
```

Examples

This statement makes visible the MenuItem called m_options on the menu m_status:

```
m_status.m_options.Show( )
```

This statement makes the child window w_child visible:

```
w_child.Show( )
```

See also

Hide

ShowHelp

Description

Provides access to a Microsoft Windows 3.x-based Help system that you have created for your PowerBuilder application. When you call ShowHelp, PowerBuilder starts the Windows help executable and displays the Help file you specify.

Syntax

ShowHelp (*helpfile, helpcommand* {, *typeid* })

Parameter	Description
helpfile	A string whose value is the name of the file that contains the compiled Help file (the .HLP file)
helpcommand	A value of the HelpCommand enumerated type. Values are: ♦ Index! — Displays the top-level contents topic in the Help file; *typeid* is ignored. ♦ Keyword! — Goes to the topic identified by the keyword in *typeid*. ♦ Topic! — Displays the topic identified by the number in *typeid*.
typeid (optional)	A number identifying the topic if *helpcommand* is Topic! or a string whose value is a keyword of a help topic if *helpcommand* is Keyword!.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

To provide context-sensitive help, use ShowHelp in appropriate scripts throughout your application with specific topic IDs or keywords.

If you specify Keyword! for *helpcommand* and the string in *typeid* is not unique, the Help Search window displays.

For information on how to create help files, see your Windows documentation.

Examples

This statement displays the Help index in the INQ.HLP file:

```
ShowHelp("C:\PB\INQ.HLP", Index!)
```

This statement displays Help topic 143 in the file EMP.HLP file:

```
ShowHelp("EMP.HLP", Topic!, 143)
```

This statement displays the Help topic associated with the keyword Part# in the file EMP.HLP:

```
ShowHelp("EMP.HLP", Keyword!, "Part#")
```

This statement displays the Help search window. The word in the box above the keyword list is the first keyword that begins with M:

```
ShowHelp("EMP.HLP", Keyword!, "M")
```

Sign

Description

Determines the sign of a number. Sign reports whether a number is negative, zero, or positive.

Syntax

Sign (*n*)

Parameter	Description
n	The number for which you want to determine the sign

Return value

Integer. Returns a number (-1, 0, or 1) indicating the sign of *n*.

Examples

This statement returns 1 (the number is positive):

```
Sign(5)
```

This statement returns 0 (zero has no sign):

```
Sign(0)
```

This statement returns -1 (the number is negative):

```
Sign(-5)
```

See also

Sign in Chapter 2, "DataWindow Painter Functions"

SignalError

Description

Causes a SystemError event at the application level.

Syntax

SignalError ( )

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs. The return value is usually not used.

Usage

Use SignalError to test error-processing scripts during development. To do so, assign values to fields in the Error object and then trigger a SystemError event. You can examine how the SystemError event script handles the forced error.

SignalError can also be useful in an actual application. For example, if a user error is so severe that you do not want the application to continue, you can set values in the Error object, including your own error number, and call SignalError. You will need to include code in the SystemError event script to recognize and handle the error you've created.

If there is no script for the SystemError event, the SignalError function does nothing.

For a list of execution-time error numbers that will be assigned to the Number attribute of the Error object when an application error occurs, see the *User's Guide.*

Example

The following statements set values in the Error object and then trigger a SystemError event so the error processing for these values can be tested:

```
Error.Number = 101
Error.Text = "Salary must be a positive number."
Error.Windowmenu = "w_emp"
Error.Object = "cb_process"
Error.ObjectEvent = "clicked"
Error.Line = 48
SignalError( )
```

Sin

Description

Calculates the sine of an angle.

Syntax

Sin (*n*)

Parameter	Description
n	The angle (in radians) for which you want the sine

Return value

Double. Returns the sine of *n*.

Examples

This statement returns .8414709848078965:

```
Sin(1)
```

This statement returns 0:

```
Sin(0)
```

This statement returns 0:

```
Sin(Pi(1))
```

See also

Cos
Pi
Tan
Sin in Chapter 2, "DataWindow Painter Functions

Sort

Description

Sorts the rows in a DataWindow control using the DataWindow's current sort criteria.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Sort** ( )

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow in which you want to sort the rows

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

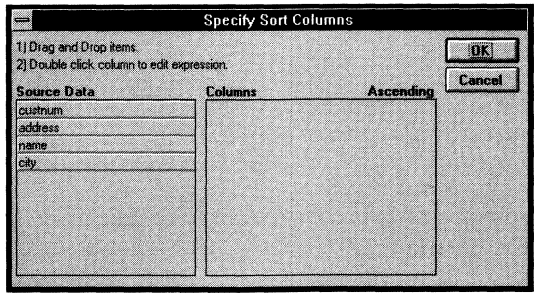
Usage

Sort uses the current sort criteria for the DataWindow. To change the sort criteria, use the SetSort function. The SetSort function is equivalent to using the Sort command on the Rows menu of the DataWindow painter. If you do not call SetSort to set the sort criteria before you call Sort, Sort uses the sort criteria specified in the DataWindow object definition.

When the Retrieve function retrieves data for the DataWindow, PowerBuilder applies the sort criteria that was defined for the DataWindow object, if any. You only need to call Sort after you change the sort criteria with SetSort or if the data has changed because of processing or user input.

> **Sorting and groups**
> To sort a DataWindow object with groups, call GroupCalc after you call Sort.

To let the user specify the sort criteria, pass a null string to the SetSort function before you call Sort. When you call Sort, PowerBuilder displays the Specify Sort Columns dialog with the sort specifications blank.



Examples

This example sets dw_employee to be sorted by column 1 ascending and then by column 2 descending. Then it sorts the rows:

```
dw_employee.SetSort("#1 A, #2 D")
dw_Employee.Sort( )
```

In this example, the rows in the DataWindow dw_depts are grouped based on department and the rows are sorted based on employee name. If the user has changed the department of several employees, then the following commands apply the sort criteria so that each group is in alphabetical order and regroup the rows:

```
dw_depts.Sort( )
dw_depts.GroupCalc( )
```

The following example causes PowerBuilder to display the Specify Sort Columns dialog because the sort criteria have been set to null:

```
string null_str
SetNull(null_str)
dw_main.SetSort(null_str)
dw_main.Sort( )
```

See also

GroupCalc
SetSort

Space

Description

Builds a string of the specified length whose value consists of spaces.

Syntax

Space (*n*)

Parameter	Description
n	A long whose value is the length of the string you want filled with spaces. The maximum value is 60000, which is the maximum size for strings.

Return value

String. Returns a string filled with *n* spaces if it succeeds and the empty string ("") if an error occurs.

Examples

This statement puts a string whose value is 4 spaces in Name:

```
string Name
Name = Space(4)
```

This statement assigns 40 spaces to the string Name:

```
string Name
Name = Space(40)
```

See also

Fill
Space in Chapter 2, "DataWindow Painter Functions"

Sqrt

Description

Calculates the square root of a number.

Syntax

Sqrt (*n*)

Parameter	Description
n	The number for which you want the square root

Return value

Double. Returns the square root of *n*.

Usage

Sqrt(*n*) is the same as *n*^.5.

Taking the square root of a negative number causes an execution error.

Examples

This statement returns 1.414213562373095:

```
Sqrt(2)
```

This statement results in an error at execution time:

```
Sqrt(-2)
```

See also

Sqrt in Chapter 2, "DataWindow Painter Functions"

Start

Description

Executes a pipeline object, which transfers data from the source to the destination as specified by the SQL query in the pipeline object. This pipeline object is an attribute of a user object inherited from the pipeline system object.

Applies to

Pipeline objects

Syntax

pipelineobject.**Start** (*sourcetrans*, *destinationtrans*, & *errordatawindow* {, *arg1*, *arg2*,..., *argn* })

Parameter	Description
pipelineobject	The name of a pipeline user object that contains the pipeline object to be executed.
sourcetrans	The name of a transaction object with which to connect to the source database.
destinationtrans	The name of a transaction object with which to connect to the target database.
errordatawindow	The name of a DataWindow control in which to display the pipeline-error DataWindow. You don't need to assign a DataWindow object to the DataWindow control. If you do it, will be replaced with the one used by the pipeline.
argn	One or more retrieval arguments as specified for the pipeline object in the Data Pipeline painter.

Return value

Integer. Returns *1* if it succeeds and a negative number if an error occurs. Error values are:

- *-1* Pipe open failed
- *-2* Too many columns
- *-3* Table already exists
- *-4* Table does not exist
- *-5* Missing connection
- *-6* Wrong arguments
- *-7* Column mismatch

- *-8* Fatal SQL error in source
- *-9* Fatal SQL error in destination
- *-10* Maximum number of errors exceeded
- *-12* Bad table syntax
- *-13* Key required but not supplied
- *-15* Pipe already in progress
- *-16* Error in source database
- *-17* Error in destination database
- *-18* Destination database is read-only

Usage

A pipeline transfer involves several PowerBuilder objects. You need:

- A pipeline object, which you define in the Data Pipeline painter. It contains the SQL statements that specify what data is transferred and how that data is mapped from the tables in the source database to those in the target database.
- A user object inherited from the pipeline system object. It inherits attributes that let you check the progress of the pipeline transfer. In the painter, you define instance variables and write scripts for pipeline events.
- A window that contains a DataWindow control for the pipeline-error DataWindow. Do not put a DataWindow object in the control. The control will display PowerBuilder's pipeline-error DataWindow object if errors occur when the pipeline executes.

  The window can also include buttons, MenuItems, or some other means to execute the pipeline, repair errors, and cancel the execution. The scripts for these actions will use the functions Start, Repair, and Cancel.

Before the application executes the pipeline, it needs to connect to the source and destination databases, create an instance of the user object, and assign the pipeline object to the user object's DataObject attribute. Then it can call Start to execute the pipeline. This code may be in one or several scripts.

When you execute the pipeline, the piped data is committed according to the settings you make in the Data Pipeline painter. You can specify that:

- The data is committed when the pipeline finishes. If the maximum error limit is exceeded, all data is rolled back.
- Data is committed at regular intervals, after a specified number of rows have been transferred. When the maximum error limit is exceeded, all rows already transferred are committed.

For information about specifying the pipeline object in the Data Pipeline painter and how the settings affect committing, see the *User's Guide.* For more information on using a pipeline in an application, see *Building Applications.*

When you dynamically assign the pipeline object to the user object's DataObject attribute, you must remember to include the pipeline object in a dynamic library when you build your application's executable.

Example

The following script creates an instance of the pipeline user object, assigns a pipeline object to the pipeline user object's DataObject attribute and executes the pipeline. I_src and i_dst are transaction objects that have been previously declared and created. Another script has established the database connections.

U_pipe is the user object inherited from the pipeline system object. I_upipe is an instance variable of type u_pipe. P_pipe is a pipeline object created in the Data Pipeline painter:

```
i_upipe = CREATE u_pipe
i_upipe.DataObject = "p_pipe"
i_upipe.Start(i_src, i_dst, dw_1)
```

See also

Cancel
Repair

StartHotLink

Description

Establishes a hot link with a DDE server application so that PowerBuilder will be notified immediately of any changes in the specified data. When the data changes in the server application, it triggers a HotLinkAlarm event in the current application.

Platform information
This and other DDE functions have no effect on the Macintosh.

Syntax

StartHotLink (*location, applname, topic*)

Parameter	Description
location	A string whose value is the location of the data in which a change of value will trigger a HotLinkAlarm event. The format of the location depends on the application that contains the data.
applname	A string whose value is the DDE name of the server application.
topic	A string identifying the data or the instance of the application in which a change will trigger a HotLinkAlarm event (for example, in Microsoft Excel, the topic name could be the name of an open spreadsheet).

Return value

Integer. Returns *1* if it succeeds. If an error occurs, StartHotLink returns a negative integer. Values are:

- *-1* No server
- *-2* Request denied

Usage

After establishing a hot link, you can include the following functions in the HotLinkAlarm event:

- GetDataDDEOrigin — To determine what application sent the notification of changed data
- GetDataDDE — To obtain the new data
- RespondRemote — To acknowledge receipt of the data

Examples

In this example, another PowerBuilder application has called the StartServerDDE function and identified itself as MyPBApp. This statement in your application establishes a hot link to data in MyPBApp. The values you specify for *location* and *topic* depend on conventions established by MyPBApp:

```
StartHotLink("Any", "MyPBApp", "Any")
```

This statement establishes a hot link with Microsoft Excel, which will notify the PowerBuilder window when the data at row 1 column 2 of REGION.XLS changes:

```
StartHotLink("R1C2", "Excel", "Region.XLS")
```

See also StopHotLink

StartServerDDE

Description

Establishes your application as a DDE server. You specify the DDE name, topic, and items that you will support.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

StartServerDDE ({ *windowname,*} *applname*, *topic* {, *item* })

Parameter	Description
windowname (optional)	The name of the server window. The default is the current window.
applname	The DDE name for your application.
topic	A string whose value is the basic data grouping the DDE client application will reference.
item (optional)	A comma-separated list of one or more strings (data within topic) that specify what your DDE server application will support (for example, "Table1","Table2").

Return value

Integer. Returns *1* if it succeeds. If an error occurs, StartServerDDE returns *-1*, meaning the your application is already started as a server.

Usage

When a DDE client application sends a DDE request, the request includes one of the items you've declared you support. You determine how your application responds to each of those items.

A window must be open to provide a handle for the DDE conversation. You can't call StartServerDDE and other DDE functions in an application object's events.

When your application has established itself as a DDE server, other applications can send DDE requests that trigger these events in your application:

- RemoteHotLinkStart — Triggered when a client sends a request for a hot link
- RemoteExec — Triggered when a client sends a command to your application. In the event's script, you can:
  - Call GetCommandDDEOrigin to find out what client application sent the command
  - Call GetCommandDDE to obtain the command
- RemoteSend — Triggered when the client sends data. In the event's script, you can call:
  - Call GetDataDDEOrigin to find out what client application sent the data
  - Call GetDataDDE to obtain the data
- RemoteRequest — Triggered when the client application requests data from your server application. In the event's script, you can call:
  - Call SetDataDDE to send the requested data
  - Call RespondRemote to acknowledge the request
- RemoteHotLinkStop — Triggered when the client wants to terminate the hot link

Example

This statement causes your PowerBuilder application to begin acting as a server. It is known to other DDE applications as MyPBApp; its topic is System, and it supports items called Table1 and Table2:

```
StartServerDDE(w_emp, "MyPBApp","System",  &
   "Table1", "Table2")
```

See also

StopServerDDE

State

Description

Determines whether an item in a ListBox control is highlighted.

Applies to

ListBox controls

Syntax

listboxname.**State** (*index*)

Parameter	Description
listboxname	The name of the ListBox in which you want to obtain the state (highlighted or not highlighted) of the item identified by *index*
index	The number of the item for which you want to obtain the state

Return value

Integer. Returns *1* if the item in *listboxname* identified by *index* is highlighted and *0* if it is not. If the index does not point to a valid item number, State returns *-1*.

Usage

The State and SetState functions are meant for a ListBox that allows multiple selections (its MultiSelect attribute is TRUE). To find all of a list's selected items, loop through the list, checking the state of each item.

The SelectedItem and SelectItem functions are meant for single-selection ListBoxes. SelectedItem reports the selection directly with no need for looping. In a multiple-selection ListBox, SelectedItem reports the first selected item only.

When you know the index of an item, you can use the Text function to get the item's text.

Examples

If item 3 in lb_Contact is selected (highlighted), then this example sets li_Item to 1:

```
integer li_Item
li_Item = lb_Contact.State(3)
```

The following statements obtain the text of all the selected items in a ListBox that allows the user to select more than one item. The MessageBox function displays each item as it is found. You could include other processing that created an array or list of the selected values:

```
integer li_ItemTotal, li_ItemCount

// Get the number of items in the ListBox.
li_ItemTotal = lb_contact.TotalItems( )

// Loop through all the items.
FOR li_ItemCount = 1 to li_ItemTotal
   // Is the item selected?  If so, display the text
   IF lb_Contact.State(li_ItemCount) = 1 THEN  &
      MessageBox("Selected Item",  &
      lb_Contact.text(li_ItemCount))
NEXT
```

This statement executes some statements if item 3 in the ListBox lb_Contact is highlighted:

```
IF lb_Contact.State(3) = 1 THEN . . .
```

See also

SelectedItem
SetState

StopHotLink

Description

Terminates a hot link with a DDE server application.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

StopHotLink (*location, applname, topic*)

Parameter	Description
location	A string whose value is the location at which you want to end the hot link, as specified in the StartHotLink function that established the link
applname	A string whose value is the DDE name of the server application, as specified in the StartHotLink function
topic	A string identifying the data or the instance of the application in which the hot link will be stopped, as specified in the StartHotLink function

> **Caution**
> *All arguments must match the arguments in an earlier StartHotLink call.*

Return value

Integer. Returns *1* if it succeeds. If an error occurs, StopHotLink returns a negative integer. Values are:

- *-1* Link was not started
- *-2* Request denied
- *-3* Could not terminate server

Examples

If another PowerBuilder application called StartServerDDE to establish itself as a server using the name MyPBApp, then your application can act as a DDE client and call StartHotLink to establish a hot link with MyPBApp. The following statement ends that hot link. The values you specify for *location* and *topic* depend on conventions established by MyPBApp:

```
StopHotLink("Any", "MyPBApp", "Any")
```

This statement stops the hot link with Microsoft Excel for row 1 column 2 in the spreadsheet REGION.XLS:

```
StopHotLink("R1C2", "Excel", "Region.XLS")
```

See also

StartHotLink

StopServerDDE

Description

Causes your application to stop acting as a DDE server application. *Any subsequent requests* from a DDE client application will fail.

> **Platform information**
> This and other DDE functions have no effect on the Macintosh.

Syntax

StopServerDDE ({ *windowname,* } *applname, topic*)

Parameter	Description
windowname (optional)	The name of the server window. The default is the current window. If you have more than one server window, *windowname* is required.
applname	The DDE name for your PowerBuilder application.
topic	A string whose value is the topic you declared when you called StartServerDDE.

Return value

Integer. Returns *1* if it succeeds. If an error occurs, StopServerDDE returns *-1*, meaning the DDE server was not started.

> ***Caution***
> *The arguments applname and topic must match the arguments in a prior StartServerDDE call.*

Example

This statement causes the PowerBuilder application MyPBApp to stop acting as a server:

```
StopServerDDE(w_emp, "MyPBApp", "System")
```

See also

StartServerDDE

String

Description

String has two syntaxes:

- Syntax 1 formats data as a string according to a specified display format mask. You can convert and format date, DateTime, numeric, and time data. You can also apply a display format to a string.
- Syntax 2 converts a blob to a string.

Syntax 1

String (*data*, { *format* })

Parameter	Description
data	The data you want returned as a string with the specified formatting. *Data* can have a date, DateTime, numeric, time, or string data type.
format (optional)	A string whose value is the display masks you want to use to format the data. The masks consists of formatting information specific to the data type of *data*. If *data* is type string, *format* is required. The format can consist of more than one mask, depending on the data type of *data*. Each mask is separated by a semicolon. See Usage for details on each data type.

Return value 1

String. Returns *data* in the specified format if it succeeds and the empty string ("") if the data type of *data* does not match the type of display mask specified or *format* is not a valid mask.

Syntax 2

String (*blob*)

Parameter	Description
blob	The blob whose value you want returned as a string

Return value 2

String. Returns the value of *blob* as a string if it succeeds and the empty string ("") if it fails. It the blob does not contain string data, String interprets the data as characters, if possible, and returns a string.

Usage

For date, DateTime, numeric, and time data, PowerBuilder uses the system's default format for the returned string if you don't specify a format. For numeric data, the default format is the [General] format.

For string data, a display format mask is required. (Otherwise, the function would have nothing to do.)

The format can consist of one or more masks:

- Formats for date, DateTime, string, and time data can include one or two masks. The first mask is the format for the data; the second mask is the format for a null value.

- Formats for numeric data can have up to four masks. A format with a single mask handles both positive and negative data. If there are additional masks, the first mask is for positive values, and the additional masks are for negative, zero, and NULL values.

For more information on specifying display formats, see Chapter 3, "Display Formats." Note that, although a format can include color specifications, they have no effect when you use String in PowerScript. Colors appear only for display formats specified in the DataWindow painter.

If the display format doesn't match the data type, PowerBuilder will try to apply the mask, which can produce unpredictable results.

You can also use String to extract a string from the Message object after calling TriggerEvent or PostEvent. See those events for more information.

Examples

This statement applies a display format to a date value and returns Jan 31, 1998:

```
String(1998-01-31, "mmm dd, yyyy")
```

This example applies a format to the value in order_date and sets date1 to 6-11-95:

```
Date order_date = 1995-06-11
string date1
date1 = String(order_date,"m-d-yy")
```

This example includes a format for a NULL date value so that when order_date is NULL, date1 is set to none:

```
Date order_date = 1995-06-11
string date1
SetNull(order_date)
date1 = String(order_date,"m-d-yy;'none'")
```

This statement applies a format to a DateTime value and returns Jan 31, 1998 6 hrs and 8 min:

```
String(DateTime(1998-01-31, 06:08:00),  &
   'mmm dd, yyyy h "hrs and" m "min"')
```

This example builds a DateTime value from the system date and time using the Today and Now functions. The String function applies formatting and sets the text of sle_date to that value, for example, 6-11-95 8:06 pm:

```
DateTime sys_datetime
string datetime1
sys_datetime = DateTime(Today(), Now())
sle_date.text = String(sys_datetime,  &
      "m-d-yy h:mm am/pm;'none'")
```

This statement applies a format to a numeric value and returns $5.00:

```
String(5,"$#,##0.00")
```

These statements set string1 to 0123:

```
integer nbr = 123
string string1
string1 = String(nbr,"0000;(000);****;empty")
```

These statements set string1 to (123):

```
integer nbr = -123
string string1
string1 = String(nbr,"000;(000);****;empty")
```

These statements set string1 to ****:

```
integer  nbr = 0
string string1
string1 = String(nbr,"0000;(000);****;empty")
```

These statements set string1 to "empty":

```
integer nbr
string string1
SetNull(nbr)
string1 = String(nbr,"0000;(000);****;empty")
```

This statement formats string data and returns A-B-C. The display format assigns a character in the source string to each @ and inserts other characters in the format at the appropriate positions:

```
String("ABC", "@-@-@")
```

This statement returns A*B:

```
String("ABC", "@*@")
```

This statement returns ABC:

```
String("ABC", "@@@")
```

This statement returns a space:

```
String("ABC", " ")
```

This statement applies a display format to time data and returns 6 hrs and 8 min:

```
String(06:08:02,'h "hrs and" m "min"')
```

This statement returns 08:06:04 pm:

```
String(20:06:04,"hh:mm:ss am/pm")
```

This statement returns 8:06:04 am:

```
String(08:06:04,"h:mm:ss am/pm")
```

Syntax 2

This example converts the blob instance variable ib_sblob, which contains string data, to a string and stores the result in sstr:

```
string sstr
sstr = String(ib_sblob)
```

This example stores today's date and test status information in the blob bb. Pos1 and pos2 store the beginning and end of the status text in the blob. Finally, BlobMid extracts a "sub-blob" that String converts to a string. Sle_status displays the returned status text:

```
blob{100} bb
long pos1, pos2
string test_status
date test_date

test_date = Today()
IF DayName(test_date) = "Wednesday" THEN  &
   test_status = "Coolant Test"
IF DayName(test_date) = "Thursday" THEN  &
   test_status = "Emissions Test"

// Store data in the blob
pos1 = BlobEdit( bb, 1, test_date)
pos2 = BlobEdit( bb, pos1, test_status )

. . . // Some processing

// Extract the status stored in bb and display it
sle_status.text = String(  &
   BlobMid(bb, pos1, pos2 - pos1))
```

See also

String in Chapter 2, "DataWindow Painter Functions"

SyntaxFromSQL

Description

Generates DataWindow source code based on a SQL SELECT statement.

Applies to

Transaction objects

Syntax

transaction.**SyntaxFromSQL** (*sqlselect*, *presentation*, *err*)

Parameter	Description
transaction	The name of a connected transaction object.
sqlselect	A string whose value is a valid SQL SELECT statement.
presentation	A string whose value is the default presentation style you want for the DataWindow. The simple format is: `Style(Type=presentationstyle)` Values for *presentationstyle* correspond to the styles in the New DataWindow dialog in the DataWindow painter. Keywords are: ♦ (Default) Tabular ♦ Grid ♦ Form (for freeform) ♦ Crosstab ♦ Graph ♦ Group ♦ Label ♦ Nup See Usage for a list of all the keywords you can use in *presentation*.
err	A string variable to which PowerBuilder will assign any error messages that occur.

Return value

String. Returns the empty string ("") if an error occurs. If SyntaxFromSQL fails, *err* may contain error messages if warnings or soft errors occur (for example, a syntax error).

Usage

To create a DataWindow object, you can pass the source code returned by SyntaxFromSQL directly to the Create function.

> **Note for SQL Server**
> If your DBMS is SQL Server and you call SyntaxFromSQL when transaction processing is on, PowerBuilder cannot determine whether the indexes are updatable and assumes they are not. Therefore, you should set AutoCommit to TRUE before you call SyntaxFromSQL.

The *presentation* string can also specify object keywords followed by attributes and values to customize the DataWindow. You can specify the style of a column, the entire DataWindow, areas of the DataWindow, and text in the DataWindow. The object keywords are:

- Column
- DataWindow
- Group
- Style
- Text
- Title

A full presentation string has the format:

```
"Style(Type=value attribute=value ...) &
DataWindow(attribute=value ...) &
Column(attribute=value ...) &
Group(groupby_col1 groupby_col2 ... attribute ...) &
Text(attribute=value ...) &
Title('titlestring')"
```

The checklists at the beginning of Appendix A identify the attributes that you can use for each object keyword.

Examples

The following statements display the DataWindow source for a tabular DataWindow object generated by the SyntaxFromSQL function in a MultiLineEdit. If errors occur, PowerBuilder fills the string ERRORS with any error messages that are generated:

```
string ERRORS, sql_syntax

sql_syntax = "SELECT emp_data.emp_id," &
   + "emp_data.emp_name FROM emp_data " &
   + "WHERE emp_data.emp_salary >45000"

mle_sql.text = &
   SQLCA.SyntaxFromSQL(sql_syntax, "", ERRORS)
```

The following statements create a grid DataWindow dw_1 from the DataWindow source generated in the SyntaxFromSQL function. If errors occur, the string ERRORS will contain any error messages that are generated, which are displayed to the user in a message box. Note that you need to call SetTransObject with SQLCA as its argument before you can call the Retrieve function:

```
string ERRORS, sql_syntax
string presentation_str, dwsyntax_str

sql_syntax = "SELECT emp_data.emp_id,"&
   + "emp_data.emp_name FROM emp_data "&
   + "WHERE emp_data.emp_salary > 45000"
```

```
presentation_str = "style(type=grid)"

dwsyntax_str = SQLCA.SyntaxFromSQL(sql_syntax,  &
   presentation_str, ERRORS)

IF Len(ERRORS) > 0 THEN
   MessageBox("Caution",  &
   "SyntaxFromSQL caused these errors: " + ERRORS)
   RETURN
END IF

dw_1.Create( dwsyntax_str, ERRORS)

IF Len(ERRORS) > 0 THEN
   MessageBox("Caution",  &
   "Create cause these errors: " + ERRORS)
   RETURN
END IF
```

See also

Create
Appendix A, "DataWindow Object Attributes"

Tan

Description

Calculates the tangent of an angle.

Syntax

Tan (*n*)

Parameter	Description
n	The angle (in radians) for which you want the tangent

Return value

Double. Returns the tangent of *n* if it succeeds and *-1* if an error occurs.

Examples

Both these statements return 0:

```
Tan(0)
Tan(Pi(1))
```

This statement returns 1.55741:

```
Tan(1)
```

See also

Cos

Pi
Sin
Tan in Chapter 2, "DataWindow Painter Functions"

Text

Description

Obtains the text of an item in a ListBox or DropDownListBox.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**Text** (*index*)

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want the text of an item
index	The number of the item for which you want the text

Return value

String. Returns the text of the item in *listboxname* identified by *index*. If the index does not point to a valid item number, Text returns the empty string ("").

Example

Assume the ListBox lb_Cities contains:

Atlanta
Boston
Chicago
Denver

Then these statements store the text of item 3, which is Chicago, in current_city:

```
string current_city
current_city = lb_Cities.Text(3)
```

See also

FindItem
SelectedItem
SelectedText

TextLine

Description

Obtains the text of the line that contains the insertion point. TextLine works for controls that can contain multiple lines.

Applies to

DataWindow, EditMask, and MultiLineEdit controls

Syntax

editname.**TextLine** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, or MultiLineEdit in which you want the text on the line that contains the insertion point

Return value

String. Returns the text on the line with the insertion point in *editname*. If an error occurs, TextLine returns the empty string ("").

Usage

If *editname* is a DataWindow control, then TextLine reports information about the edit control over the current row and column.

Examples

In the MultiLineEdit mle_state, if the insertion point is on line 4 and its text is North Carolina, then this example sets linetext to North Carolina:

```
string linetext
linetext = mle_state.TextLine( )
```

If the insertion point is on a line whose text is Y in the MultiLineEdit mle_contact, then some processing takes place:

```
IF mle_contact.TextLine( ) = "Y" THEN . . .
```

See also

SelectedItem

Time

Description

Converts DateTime, string, or numeric data to data of type time. It also extracts a time value from a blob. You can use one of three syntaxes, depending on the data type of the source data:

- To extract the time from DateTime data, or to extract a time stored in a blob, use Syntax 1.
- To convert a string to a time, use Syntax 2.
- To combine numbers for hours, minutes, and seconds into a time value, use Syntax 3.

Syntax 1

Time (*datetime*)

Parameter	Description
datetime	A DateTime value or a blob in which the first value is a time or DateTime value. The rest of the contents of the blob is ignored.

Return value 1

Time. Returns the time in *datetime* as a time. If *datetime* does not contain a valid time, Time returns *00:00:00.000000*. If *datetime* is NULL, Time returns NULL.

Syntax 2

Time (*string*)

Parameter	Description
string	A string whose value is a valid time (such as 8AM or 10:25) that you want returned as a time. Only the hour is required; you do not have to include the minutes, seconds, or microseconds of the time or AM or PM. The default value is 00 for minutes and seconds and 000000 for microseconds. PowerBuilder determines whether the time is AM or PM based on a 24-hour clock.

Return value 2

Time. Returns the time in *string* as a time. If string does not contain a valid time, Time returns *00:00:00.000000*.

Syntax 3

Time (*hour, minute, second* {, *microsecond* })

Parameter	Description
hour	The integer for the hour (00 to 23) of the time
minute	The integer for the minutes (00 to 59) of the time
second	The integer for the seconds (0 to 59) of the time
microsecond (optional)	The integer for the microseconds (0 to 599999) of the time

Return value 3

Time. Returns the time as a time data type and *00:00:00* if the value in any argument is not valid (out of the specified range of values). If any argument is NULL, Time returns NULL.

Usage

Valid times can include any combination of hours (00 to 23), minutes (00 to 59), seconds (00 to 59), and microseconds (0 to 599999).

Examples

After StartDateTime has been retrieved from the database, this example sets StartTime equal to the time in StartDateTime:

```
DateTime StartDateTime
time StartTime
. . .
StartTime = Time(StartDateTime)
```

Suppose that the value of a blob variable ib_blob contains a DateTime value beginning at byte 32. The following statement extracts the time from the value:

```
time lt_time
lt_time = Time(BlobMid(ib_blob, 32))
```

Syntax 2

These statements set What_Time to NULL:

```
Time What_Time
string null_string

SetNull(null_string)
What_Time = Time(null_string)
```

This statement returns a time value for 45 seconds before midnight (23:59:15), which is specified as a string:

```
Time("23:59:15")
```

This statement converts the text in the SingleLineEdit sle_Time_Received to a time value:

```
Time(sle_Time_Received.Text)
```

Syntax 3

These statements set What_Time to 10:15:45:234 and display the resulting time as a string in st_1. The default display format doesn't include microseconds, so the String function specifies a display format with microseconds:

```
Time What_Time
What_Time = Time(10, 15, 45, 234)
st_1.Text = String(What_Time, "hh:mm:ss:ffffff")
```

These statements set What_Time to 10:15:45:

```
Time What_Time
What_Time = Time(10, 15, 45)
```

See also

Time in Chapter 2, "DataWindow Painter Functions"

Timer

Description

Repeatedly triggers a Timer event in a window at the specified interval. When you call Timer, it starts a timer. When the interval is over, PowerBuilder triggers the Timer event and resets the timer.

Syntax

Timer (*number* {, *windowname* })

Parameter	Description
interval	The number of seconds (0-65) that you want between timer events. If *interval* is 0, Timer turns off the timer so that it no longer triggers Timer events.
windowname (optional)	The window in which you want the timer event to be triggered. The window must be an open window. If you do not specify a window, the timer event occurs in the current window.

Return value

Integer. Returns *1* if succeeds and *-1* if an error occurs.

Timers in Microsoft Windows
When you select a number between 0 and .055 (about 1/18 of a second), the timer event is triggered at approximately .055 seconds, the finest granularity the Windows system allows.

Microsoft Windows 3.x supports up to 16 concurrent timers in the system.

Examples

This statement triggers a timer event every 2 seconds in the active window:

```
Timer(2)
```

This statement stops the triggering of the timer event in the active window:

```
Timer(0)
```

These statements trigger a timer event every half second in the window w_Train:

```
Open(w_Train)
Timer(0.5, w_Train)
```

This example causes the current time to be displayed in a StaticText control in a window. Calling Timer in the window's Open event script starts the timer. The script for the Timer event refreshes the displayed time.

In the window's Open event script, the following code displays the time initially and starts the timer:

```
st_time.Text = String(Now(), "hh:mm")
Timer(60)
```

In the window's Timer event, which is triggered every minute, this code displays the current time in the StaticText st_time:

```
st_time.Text = String(Now(), "hh:mm")
```

See also

Idle

Today

Description

Obtains the system date.

Syntax **Today ( )**

Return value Date. Returns the current system date.

Examples This statement returns the current system date:

```
Today( )
```

This statement executes some statements when the current system date is before April 15, 1995:

```
IF Today( ) < 1995-04-15 THEN . . .
```

This statement displays the current date in the StaticText st_date in the corner of a window:

```
st_date.Text = String(Today( ), "m/d/yy")
```

See also Now
Today in Chapter 2, "DataWindow Painter Functions"

Top

Description Obtains the index number of the first visible item in a ListBox. Top lets you to find out how the user has scrolled the list.

Applies to ListBox controls

Syntax *listboxname*.**Top ( )**

Parameter	Description
listboxname	The name of the ListBox in which you want the index of the first visible item in the list

Return value Integer. Returns the index of the first visible item in *listboxname*. Top returns *-1* if an error occurs.

Usage

The index of a list item is its position in the full list of items, regardless of how many are currently visible in the control.

Examples

If item 15 has been scrolled to the top of the list in lb_Contacts, then this example sets Num to 15:

```
integer Num
Num = lb_Contacts.Top( )
```

If the user has not scrolled the list in lb_Contacts, then Num is set to 1:

```
integer Num
Num = lb_Contacts.Top( )
```

If the item at the top of the list in lb_Contacts is not the currently selected item, the following statements scroll the currently selected item to the top:

```
integer Num
Num = lb_Contacts.SelectedIndex( )
IF lb_Contacts.Top( ) <> Num THEN  &
   lb_contacts.SetTop(Num)
```

See also

SelectedIndex
SetTop

TotalItems

Description

Determines the total number of items in a ListBox or DropDownListBox.

Applies to

ListBox and DropDownListBox controls

Syntax

listboxname.**TotalItems** ( )

Parameter	Description
listboxname	The name of the ListBox or DropDownListBox in which you want the total number of items

Return value

Integer. Returns the total number of items in *listboxname*. If *listboxname* contains no items, TotalItems returns *0*. If an error occurs, it returns *-1*.

Examples

If lb_Actions contains a total of 5 items, this example sets Total to 5:

```
integer Total
Total = lbx_Actions.TotalItems( )
```

This FOR loop is executed for each item in lb_Actions:

```
integer Total, n
Total = lb_Actions.TotalItems( )
FOR n = 1 to Total
   . . .  // Some processing
NEXT
```

See also

TotalSelected

TotalSelected

Description

Determines the number of items in a ListBox that are selected.

Applies to

ListBox controls

Syntax

listboxname.**TotalSelected** ( )

Parameter	Description
listboxname	The name of the ListBox in which you want the number of items that are selected

Return value

Integer. Returns the number of items in *listboxname* that are selected. If no items in *listboxname* are selected, TotalSelected returns *0*. If an error occurs, it returns *-1*.

Usage

TotalSelected works only if the MultiSelect attribute of *listboxname* is TRUE.

Examples

If three items are selected in lb_Actions, this example sets SelectedTotal to 3:

```
integer SelectedTotal
SelectedTotal = lb_Actions.TotalSelected( )
```

These statements in the SelectionChanged event of lb_Actions display a MessageBox if the user tries to select more than three items:

```
IF lb_Actions.TotalSelected( ) > 3 THEN
   MessageBox("Warning",  &
   "You can only select 3 items!")
ELSE
   . . .  // Some processing
END IF
```

See also TotalItems

TriggerEvent

Description

Triggers an event associated with the specified object, which executes the script for that event immediately.

Syntax

objectname.**TriggerEvent** (*event* {, *word*, *long* })

Parameter	Description
objectname	The name of any PowerBuilder object or control that has events associated with it.
event	A value of the TrigEvent enumerated data type that identifies a PowerBuilder event (for example, Clicked!, Modified!, or DoubleClicked!) or a string whose value is the name of an event. The event must be a valid event for *objectname* and a script must exist for the event in *objectname*.
word (optional)	A long value to be stored in the WordParm attribute of the system's Message object. If you want to specify a value for *long*, but not *word*, enter 0. (For cross-platform compatibility, WordParm and LongParm are both longs.)
long (optional)	A long value or a string that you want to store in the LongParm attribute of the system's Message object. When you specify a string, a pointer to the string is stored in the LongParm attribute, which you can access with the String function (see Usage).

Return value

Integer. Returns *1* if it is successful and the event script runs and *-1* if the event is not a valid event for *objectname*, or no script exists for the event in *objectname*.

Usage

If you specify the name of an event instead of a value of the TrigEvent enumerated data type, enclose the name in double quotation marks.

> **Tip**
> It is a good idea to check the return code to determine whether TriggerEvent succeeded and, based on the result, perform the appropriate processing.

You can pass information to the event script with the *word* and *long* arguments. The information is stored in the Message object. In your script, you can reference the WordParm and LongParm fields of the Message object to access the information.

If you have specified a string for *long*, you can access it in the triggered event by using the String function with the keyword "address" as the *format* parameter. Your event script might begin as follows:

```
string PassedString
PassedString = String(Message.LongParm, "address")
```

For more information about events and when to use PostEvent and TriggerEvent, see PostEvent.

To trigger system events that are not PowerBuilder-defined events, use Post or Send, instead of PostEvent and TriggerEvent.. Although Send can send messages that trigger PowerBuilder events, as shown below, you have to know the codes for a particular message. It is easier to use the PowerBuilder functions that trigger the desired events.

Equivalent syntax

Both of the following statements click the CheckBox cb_OK. The following call to the Send function:

```
Send(Handle(Parent), 273, 0, Long(Handle(cb_OK), 0))
```

is equivalent to:

```
cb_OK.TriggerEvent(Clicked!)
```

Examples

This statement executes the script for the Clicked event in the CommandButton cb_OK immediately:

```
cb_OK.TriggerEvent(Clicked!)
```

This statement executes the script for the user-defined event cb_exit_request in the parent window:

```
Parent.TriggerEvent("cb_exit_request")
```

This statement executes the script for the Clicked event in the MenuItem m_File on the menu m_Appl:

```
m_Appl.m_File.TriggerEvent(Clicked!)
```

See also

Post
PostEvent
Send

Trim

Description

Removes leading and trailing spaces from a string.

Syntax

Trim (*string*)

Parameter	Description
string	The string you want returned with leading and trailing spaces deleted

Return value

String. Returns a copy of *string* with all leading and trailing spaces deleted if it succeeds and the empty string ("") if an error occurs.

Usage

Trim is useful for removing spaces that a user may have typed before or after newly entered data.

Examples

This statement returns BABE RUTH:

```
Trim("      BABE RUTH       ")
```

This example removes the leading and trailing spaces from the user-entered value in the SingleLineEdit sle_emp_fname and saves the value in emp_fname:

```
string emp_fname
emp_fname = Trim(sle_emp_fname.Text)
```

See also

LeftTrim
RightTrim
Trim in Chapter 2, "DataWindow Painter Functions"

Truncate

Description

Truncates a number to the specified number of decimal places.

Syntax

Truncate (*x, n*)

Parameter	Description
x	The number you want to truncate
n	The number of decimal places to which you want to truncate *x* (valid values are 0 through 18).

Return value

Decimal. Returns the result of the truncation if it succeeds and a NULL if it fails.

Examples

This statement returns 9.2:

```
Truncate(9.22, 1)
```

This statement returns 9.2:

```
Truncate(9.28, 1)
```

This statement returns 9:

```
Truncate(9.9, 0)
```

This statement returns –9.2:

```
Truncate(-9.29, 1)
```

See also

Ceiling
Int
Round
Truncate in Chapter 2, "DataWindow Painter Functions"

TypeOf

Description

Determines the type of an object or control, reported as a value of the Object enumerated data type.

Applies to

Any object

Syntax

objectname.**TypeOf** ( )

Parameter	Description
objectname	The name of the object or control for which you want the type

Return value

Object enumerated data type. Returns the type of *objectname*.

Usage

Use TypeOf to determine the type of a selected or dragged control.

Examples

If dw_Customer is a DataWindow control, this statement returns DataWindow!:

```
dw_Customer.Typeof( )
```

This example looks at the first 5 controls in the w_dept window's Control array attribute. The loop executes some statements for each control that is a CheckBox:

```
integer n
FOR n = 1 to 5
  IF w_dept.Control[n].TypeOf( ) = CheckBox! THEN
    . . .  // Some processing
  END IF
NEXT
```

This loop stores in the winobject array the type of each object in the window's Control array attribute:

```
object winobjecttype[]
integer i

FOR i = 1 to UpperBound(Control[])
   winobjecttype[i] = TypeOf(Control[i])
NEXT
```

If you don't know the type of a control passed via PowerObjectParm in the Message object, the following example assigns the passed object to a graphicobject variable, the ancestor of all the control types, and assigns the type to a variable of type object, which is the enumerated data type that TypeOf returns. The CHOOSE CASE statement can include processing for each control type that you want to handle. This code would be in the Open event for a window that was opened with OpenWithParm:

```
graphicobject stp_obj
object  type_obj

stp_obj = Message.PowerObjectParm
type_obj = stp_obj.TypeOf( )

CHOOSE CASE type_obj
   CASE DataWindow!
    MessageBox("The object"," Is a datawindow")

   CASE  SingleLineEdit!
    MessageBox("The object"," Is a sle")
   . . .  // Cases for additional object types

   CASE ELSE
    MessageBox("The object"," Is irrelevant!")
END CHOOSE
```

See also ClassName

Uncheck

Description Removes the checkmark, if any, next to an item a dropdown or cascading menu and sets the item's Checked attribute to FALSE.

Applies to Menuitems in Menu objects

Syntax *menuitem*.**Uncheck ( )**

Parameter	Description
menuitem	The fully qualified name of the MenuItem from which you want to remove the checkmark, if any. The item must be on a dropdown or cascading menu, not an item on a menu bar.

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

A checkmark next to a menu item indicates that the menu option is currently on and that the user can turn the option on and off by choosing it. For example, in the Window painter's Design menu, a checkmark is displayed next to Grid when the grid is on.

You can use Check in an item's Clicked script to mark a menu item when the user turns the option on and Uncheck to remove the check when the user turns the option off.

Equivalent syntax

You can set the object's Checked attribute instead of calling Uncheck:

menuitem.Checked = FALSE

This statement:

```
m_appl.m_view.m_grid.Checked = FALSE
```

is equivalent to:

```
m_appl.m_view.m_grid.Uncheck( )
```

Example

This statement removes the checkmark next to the m_grid MenuItem in the dropdown MenuItem m_view on the menu bar m_appl:

```
m_appl.m_view.m_grid.Uncheck( )
```

This example checks whether the m_grid MenuItem in the dropdown MenuItem m_view of the menu bar m_appl is currently checked. If so, the script unchecks the item. If it is not checked, the script checks the item:

```
IF m_appl.m_view.m_grid.Checked = TRUE THEN
   m_appl.m_view.m_grid.Uncheck( )
ELSE
   m_appl.m_view.m_grid.Check( )
END IF
```

See also

Check

Undo

Description

Cancels the last edit in an edit control, restoring the text to the content before the last change.

Applies to

DataWindow, EditMask, MultiLineEdit, and SingleLineEdit controls

Syntax

editname.**Undo** ( )

Parameter	Description
editname	The name of the DataWindow control, EditMask, MultiLineEdit, or SingleLineEdit in which you want to cancel (reverse) the last edit. For a DataWindow control, reverses the last edit in the edit control over the current row and column.

Return value

Integer. Returns *1* when it succeeds and *-1* if an error occurs.

Usage

To determine whether the last action can be canceled, call the CanUndo function.

Examples

This statement reverses the last edit in MultiLineEdit mle_Contact:

```
mle_Contact.Undo( )
```

The following statement checks to see if the last edit in the MultiLineEdit mle_Contact can be reversed, and if so reverse it:

```
IF mle_Contact.CanUndo( ) THEN mle_Contact.Undo( )
```

See also

CanUndo

UnitsToPixels

Description

Converts PowerBuilder units to pixels and reports the measurement. Because pixels are not usually square, you also specify whether to convert in the horizontal or vertical direction.

Syntax

UnitsToPixels (*units, type*)

Parameter	Description
units	An integer whose value is the number of PowerBuilder units you want to convert to pixels
type	A value of the ConvertType enumerated data type value indicating how to convert the value: ♦ XUnitsToPixels! — Convert the units in the horizontal direction ♦ YUnitsToPixels! — Convert the units in the vertical direction

Return value

Integer. Returns the converted value if it succeeds and *-1* if an error occurs.

Example

These statements convert 350 vertical PowerBuilder units to vertical pixels and set value equal to the converted value:

```
integer Value
Value = UnitsToPixels(350, YUnitsToPixels!)
```

See also

PixelsToUnits

Update

Description

Updates the database with the changes made in a DataWindow control. Update can also call AcceptText for the current row and column before it updates the database.

Applies to

DataWindow controls and child DataWindows

Syntax

datawindowname.**Update** ({ *accept* {, *resetflag* } })

Parameter	Description
datawindowname	The name of the DataWindow control or child DataWindow that contains the information you want to use to update the database

Parameter	Description
accept (optional)	A boolean value specifying whether the DataWindow control should automatically perform an AcceptText prior to performing the update: ♦ TRUE — (Default) Perform AcceptText ♦ FALSE — Do not perform AcceptText
resetflag (optional)	A boolean value specifying whether *datawindowname* should automatically reset the update flags: ♦ TRUE — (Default) Reset the flags ♦ FALSE — Do not reset the flags

Return value

Integer. Returns *1* if it succeeds and *-1* if an error occurs.

Usage

If *accept* is TRUE, the update is performed only if the data passes validation.

You *must* use the SetTrans or the SetTransObject function to specify the database connection before the Update function will execute. When you use SetTransObject, the more efficient of the two, you must do your own transaction management, which includes issuing the SQL COMMIT or ROLLBACK statement to finalize the update.

By default, Update resets the update flags after successfully completing the update. However, you can prevent the flags from being reset until you perform other validations and commit the changes. When you are satisfied with the update, call ResetUpdate to clear the flags so that items are no longer marked as modified.

Use SetTransObject when *resetflag* is FALSE
You would typically use SetTransObject, not SetTrans, to specify the DataWindow control's transaction object when you plan to update with the *resetflag* argument to FALSE. Only SetTransObject gives you control of when changes are committed.

If you want to update several tables in one DataWindow control, you can use Modify to change the Update attribute of columns in each table. To preserve the status flags of the rows and columns, set the *resetflag* argument to FALSE. Because the updates all occur in the same DataWindow control, you cannot allow the flags to be cleared until all the tables have used them. When all the updates are successfully completed and committed, you can call ResetUpdate to clear the changed flags in the DataWindow. See Modify for an example of this technique.

If you are updating multiple DataWindow controls as part of one transaction, set the *resetflag* argument to FALSE. This will prevent the DataWindow from "forgetting" which rows to update in case one of the updates fails. You can roll back, try to correct the situation, and update again. Once all of the DataWindows have been updated successfully, use COMMIT to finalize the transaction and use ResetUpdate to reset the DataWindow's status flags.

If you call Update with the *resetflag* argument set to FALSE and do not call ResetUpdate, the DataWindow will attempt to issue the same SQL statements again the next time you call Update.

> ***Caution***
> *If you call Update in an ItemChanged event, be sure to set the accept argument to FALSE to avoid an endless loop and a stack fault. Because AcceptText triggers an ItemChanged event, you cannot call it in that event (see AcceptText).*

If you call Update in the ItemChanged event, then the item's old value is updated in the database, not the newly entered value. The newly entered value in the edit control is still being validated and won't become the item value until the ItemChanged event is successfully completed. If you want to include the new value in an update in the ItemChanged event, use the appropriate SetItem function first.

Events

Update may trigger these events:

- DBError
- SQLPreview
- UpdateEnd
- UpdateStart

If AcceptText is performed, it may trigger these events:

- ItemChanged
- ItemError

Examples

This example connects to the database, specifies a transaction object for the DataWindow control with SetTransObject, and then updates the database with the changes made in dw_employee. By default, AcceptText is performed on the data in the edit control for the current row and column and the status flags are reset:

```
CONNECT USING SQLCA;
dw_employee.SetTransObject(SQLCA)
. . .  // Some processing
dw_employee.Update( )
```

This example connects to the database, specifies a transaction object for the DataWindow control with SetTransObject, and then updates the database with the changes made in dw_employee. The update resets the status flags but does not perform AcceptText before updating the database:

```
CONNECT USING SQLCA;
dw_employee.SetTransObject(SQLCA)
. . .  // Some processing
dw_Employee.Update(FALSE, TRUE)
```

As before, this example connects to the database, specifies a transaction object for the DataWindow control with SetTransObject, and then updates the database with the changes made in dw_employee. After Update is executed, the example checks the return code and depending on the success of the update, executes a COMMIT or ROLLBACK:

```
integer  rtn

CONNECT USING SQLCA;
dw_employee.SetTransObject(SQLCA)
rtn = dw_employee.Update( )

IF rtn = 1 THEN
   COMMIT USING SQLCA;
ELSE
   ROLLBACK USING SQLCA;
END IF
```

See also

AcceptText
Modify
ResetUpdate
Print
SaveAs
SetTrans
SetTransObject

Upper

Description

Converts all the characters in a string to uppercase.

Syntax

Upper (*string*)

Parameter	Description
string	The string you want to convert to uppercase letters

Return value

String. Returns *string* with lowercase letters changed to uppercase if it succeeds and the empty string ("") if an error occurs.

Example

This statement returns BABE RUTH:

```
Upper("Babe Ruth")
```

See also

Lower
Upper in Chapter 2, "DataWindow Painter Functions"

UpperBound

Description

Obtains the upper bound of a dimension of an array.

Syntax

UpperBound (*array* {, *n* })

Parameter	Description
array	The name of the array for which you want the upper bound of a dimension.
n (optional)	The number of the dimension for which you want the upper bound. The default is 1.

Return value

Integer. Returns the upper bound of dimension *n* of *array*. If *n* is greater than the number of dimensions of the array, UpperBound returns *-1*.

Usage

For variable-size arrays, memory is allocated for the array when you assign values to it. UpperBound returns the largest value that has been defined for the array in the current script. Before you assign values, the lower bound is 1 and the upper bound is 0.

For fixed arrays, whose size is specified when it is declared, UpperBound always returns the declared size.

Examples

The following statements illustrate the values UpperBound reports for fixed-size arrays and for variable-size arrays before and after memory has been allocated:

```
integer a[5]
UpperBound(a)      // Returns 5
UpperBound(a,1)    // Returns 5
UpperBound(a,2)    // Returns -1; no 2nd dimension

integer b[10,20]
UpperBound(b,1)    // Returns 10
UpperBound(b,2)    // Returns 20

integer c[ ]
UpperBound(c)      // Returns 0; no memory allocated
c[50] = 900
UpperBound(c)      // Returns 50
c[60] = 800
UpperBound(c)      // Returns 60
c[60] = 800
c[50] = 700
UpperBound(c)      // Returns 60

integer d[10 to 50]
UpperBound(d)      // Returns 50
```

This example determines the position of a menu bar item called File, and if the item has a cascading menu with an item called Update, disables the Update item. The code could be a script for a control in a window.

The code includes a rather complicated construct: Parent.Menuid.Item. Its components are:

- Parent — The parent window of the control that is running the script.
- Menuid — An attribute of a window whose value identifies the menu associated with the window.
- Item — an attribute of a menu that is an array of items in that menu. If Item is itself a dropdown or cascading menu, it has its own item array, which can be a fourth qualifier.

The script is:

```
integer i, k, tot1, tot2

// Determine how many menu bar items there are.
tot1 = UpperBound(Parent.Menuid.Item)

FOR i = 1 to tot1
   // Find the position of the File item.
   IF Parent.Menuid.Item[i].text = "File" THEN
      MessageBox("Position",  &
         "File is in Position "+ string(i))
      tot2 = UpperBound(Parent.Menuid.Item[i].Item)

      FOR k = 1 to tot2
         // Find the Update item under File.
         IF Parent.Menuid.Item[i].Item[k].Text = &
            "Update" THEN
          // Disable the Update menu option.
            Parent.Menuid.Item[i].Item[k].Disable( )
            EXIT
         END IF
      NEXT
      EXIT
   END IF
NEXT
```

See also LowerBound

WorkSpaceHeight

Description Obtains the height of the workspace within the boundaries of the specified window.

Applies to Window objects

Syntax *windowname*.**WorkSpaceHeight** ( )

Parameter	Description
windowname	The name of the window for which you want the height of the workspace area

Return value

Integer. Returns the height of the workspace area in PowerBuilder units in *windowname*. If an error occurs, WorkSpaceHeight returns *-1*.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scroll bar, if any) and the top and bottom of the window (not including the thickness of the frame or the title bar, menu bar, or horizontal scroll bar, if any).

Example

This example returns the height of the workspace area in the w_employee window:

```
Integer  Height
Height = W_employee.WorkSpaceHeight( )
```

See also

WorkSpaceWidth
WorkSpaceX
WorkSpaceY
PointerX
PointerY

WorkSpaceWidth

Description

Obtains the width of the workspace within the boundaries of the specified window.

Applies to

Window objects

Syntax

windowname.**WorkSpaceWidth** ( )

Parameter	Description
windowname	The name of the window for which you want the width of the workspace area

Return value

Integer. Returns the width of the workspace area (in PowerBuilder units) in *windowname*. If an error occurs, WorkSpaceWidth returns *-1*.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scroll bar, if any) and the top and bottom of the window (not including the thickness of the frame or the title bar, menu bar, or horizontal scroll bar, if any).

Example

This example returns the width of the workspace area in the w_employee window:

```
integer Width
Width = w_employee.WorkSpaceWidth( )
```

See also

PointerX
PointerY
WorkSpaceHeight
WorkSpaceX
WorkSpaceY

WorkSpaceX

Description

Obtains the distance between the left edge of a window's workspace and the left edge of the screen.

Applies to

Window objects

Syntax

windowname.**WorkSpaceX** ( )

Parameter	Description
windowname	The name of the window for which you want the distance between the left edge of the workspace area and the left edge of the screen

Return value

Integer. Returns the distance that the left edge of the workspace area of *windowname* is from the left edge of the screen (in PowerBuilder units). WorkSpaceX returns *-1* if an error occurs.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scroll bar, if any) and the top and bottom of the window (not including the thickness of the frame or the title bar, menu bar, or horizontal scroll bar, if any).

Example

This example returns the distance from the left edge of the screen to the left edge of the workspace area in the w_employee window:

```
integer  workx
workx = w_employee.WorkSpaceX( )
```

See also

PointerX
PointerY
WorkSpaceHeight
WorkSpaceWidth
WorkSpaceY

WorkSpaceY

Description

Obtains the distance between the top of a window's workspace and the top of the screen.

Applies to

Window objects

Syntax

windowname.**WorkSpaceY** ( )

Parameter	Description
windowname	The name of the window for which you want the distance between the top of the workspace area and the top the screen

Return value

Integer. Returns the distance that the top of the workspace area of *windowname* is from the top of the screen (in PowerBuilder units). If an error occurs, WorkSpaceY returns *-1*.

Usage

The workspace area is the area between the sides of the window (not including the thickness of the frame or the vertical scroll bar, if any) and the top and bottom of the window (not including the thickness of the frame or the title bar, menu bar, or horizontal scroll bar, if any).

Example

This example returns the distance from the top of the screen to the top of the workspace area in the w_employee window:

```
integer worky
worky = w_employee.WorkSpaceY( )
```

See also

PointerX
PointerY
WorkSpaceHeight
WorkSpaceWidth
WorkSpaceX

Write

Description

Writes data to an opened OLE stream object.

Applies to

OLEStream objects

Syntax

olestream.**Write** (*dataforstream*)

Parameter	Description
olestream	The name of an OLE stream variable that has been opened
dataforstream	A string, blob, or character array whose value you want to write to *olestream*

Return value

Integer. Returns *0* if it succeeds and one of the following negative values if an error occurs:

- *-1* Stream is not open

- *-2* Read error
- *-9* Other error

Example

This example opens an OLE object in the file MYSTUFF.OLE and assigns it to the OLEStorage object olest_stuff. Then it opens the stream called info in olest_stuff and assigns it to the stream object olestr_info. It writes the contents of the instance blob variable lb_info to the stream olestr_info. Finally, it saves the storage olest_stuff:

```
boolean lb_memexists
OLEStorage olest_stuff
OLEStream olestr_info
integer result

olest_stuff = CREATE OLEStorage
result = olest_stuff.Open("c:\ole2\mystuff.ole")
IF result <> 0 THEN RETURN

result = olestr_info.Open(olest_stuff, "info",  &
      stgReadWrite!, stgExclusive!)
IF result <> 0 THEN RETURN

result = olestr_info.Write(lb_info)
IF result = 0 THEN olest_stuff.Save()
```

See also

Open
Length
Seek
Read

Year

Description

Determines the year of a date value.

Syntax

Year (*date*)

Parameter	Description
date	The date from which you want the year

Return value

Integer. Returns an integer whose value is a 4-digit year adapted from the year portion of *date* if it succeeds and 1900 if an error occurs.

When you convert a string that has a two-digit year to a date, then PowerBuilder chooses the century, as follows. If the year is between 00 to 49, PowerBuilder assumes 20 as the first two digits; if it is between 50 and 99, PowerBuilder assumes 19.

Usage

PowerBuilder handles years from 1000 to 3000 inclusive.

If your data includes date before 1950, such as birth dates, always specify a 4-digit year so that Year and other PowerBuilder functions, such as Sort, interpret the date as intended.

Example

This statement returns 1995:

```
Year(1995-01-31)
```

See also

Day
Month
Year in Chapter 2, "DataWindow Painter Functions"

Yield

Description

Yields control to other graphic objects, including objects that are not PowerBuilder objects. Yield checks the message queue and if there are messages in the queue, it pulls them from the queue.

Syntax

Yield ( )

Return value

Boolean. Returns TRUE if it pulls messages from the message queue and FALSE if there are no messages.

Usage

Include Yield within a loop so that other processes can happen. For example, use Yield to allow end users to interrupt a loop. By yielding control, you allow the user time to click on a cancel button in another window. Then code in the loop can check whether a global variable's status has changed. You can also use Yield in a loop in which you are waiting for something to finish so that other processing can take place, in either your or some other application.

Using other applications while retrieving data
Although the user can't do other activities in a PowerBuilder application while retrieving data, you can allow them to use other applications on their system. Put Yield in the RetrieveRow event so that other applications can run during the retrieval.

Of course, Yield will make your PowerBuilder application run slower because processing time will be shared with other applications.

Examples

In this example, a second window includes a button that the user can click to interrupt the loop by setting a shared variable. When the user clicks the button, its Clicked script sets s_interrupt, which the current script can check:

```
integer n
// s_interrupt is a shared variable.
s_interrupt = FALSE
for n = 1 to 3000
   Yield( )
   IF s_interrupt THEN
      MessageBox("Debug","Interrupted!")
      s_interrupt = FALSE
      EXIT
   ELSE
   . . .  // Some processing
   END IF
NEXT
```

In this example, a script wants to open a DDE channel with Lotus Notes, whose executable name is stored in the variable mailprogram. If the program isn't running, the script starts it and loops, waiting until the program's startup is finished and it can establish a DDE channel. The loop includes Yield, so that the computer can spend time actually starting the other program:

```
time starttime
long hndl

SetPointer(HourGlass!)
//Try to establish a handle; SendMail is the topic.
hndl = OpenChannel("Notes","SendMail")

//If the program is not running, start it
If hndl < 1 then
   Run(mailprogram, Minimized!)
   starttime = Now()
```

```
    // Wait up to 2 minutes for Notes to load
    // and the user to log on.
    DO
       //Yield control occasionally.
       Yield( )
       //Is Notes active yet?
       hndl = OpenChannel("Notes","SendMail")
       // If Notes is active.
       IF hndl > 0 THEN EXIT
    LOOP Until SecondsAfter(StartTime,Now()) > 120

    // If 2 minutes pass without opening a channel
    IF hndl < 1 THEN
       MessageBox("Error",  &
          "Can't start Notes.", StopSign!)
       SetPointer(Arrow!)
       RETURN
    END IF
END IF
```

CHAPTER 2

DataWindow Painter Functions

This chapter provides syntax, descriptions, and examples of the functions you can use in expressions in the DataWindow painter. The functions are listed alphabetically after a short introduction that describes where you can use these functions.

Using DataWindow Painter Functions

You can use the functions in this chapter in expressions for computed fields, filters, validation rules, and graphed data, with some exceptions. When you use the DataWindow or Report painter, the dialogs in which you define these expressions include a listbox that lists the available functions and their arguments. The dialogs make it easy to insert the function into the expression.

Return values for functions and expressions

DataWindow painter expressions can return double, string, DateTime, or time data types. Within an expression, a function can return other types, such as boolean, date, or integer. The final value of the expression will be converted to the DataWindow data types listed above.

Using aggregate functions

Aggregate functions have some restrictions. Aggregate functions can't be used in filters and validation rules. You cannot nest aggregate functions; that is, one aggregate function cannot be an argument for another aggregate function. Aggregate functions are those functions that consider a range of values in a column, such as Avg, Max, or StDev.

User-defined functions

You can include user-defined functions in DataWindow painter expressions. The data type of the function's return value can be any of the following: double, string, boolean, date, DateTime, or time. The function must be defined as a global function so that it is available to the DataWindow object.

Formatting for numbers

So that an application you build will run the same no matter in which country it is deployed, masks used in display formats and edit masks and DataWindow expressions require U.S. notation for numbers. That is, when you specify a number or number mask, comma always represents the thousands delimiter and period always represents the decimal place. During execution, the locally correct symbols are displayed for numbers. For example, in countries where comma represents the decimal place and period represents thousands, users will see numbers in those formats during execution.

Abs

Description

Calculates the absolute value of a number.

Syntax

Abs (*n*)

Parameter	Description
n	The number for which you want the absolute value

Return value

The data type of *n*. Returns the absolute value of *n*.

Example

This expression, typed on a single line, counts all the product numbers where the absolute value of the product number is distinct:

```
Count( product_number for All DISTINCT
Abs(product_number))
```

Only data with an absolute value greater than 5 passes this validation rule:

```
Abs(value_set) > 5
```

See also

Count
Abs in Chapter 1, "PowerScript Functions"

Asc

Description

Converts the first character of a string to its ASCII integer value.

Syntax

Asc (*string*)

Parameter	Description
string	The string for which you want the ASCII value of the first character

Return value

Integer. Returns the ASCII value of the first character in *string*.

Usage

Use Asc to test the case of a character or manipulate text and letters.

To find out the case of a character, you can check whether its ASCII value is within the appropriate range.

Examples

This expression for a computed field returns the string in code_id if the ASCII value of the first character in code_id is A (65):

```
If (Asc(code_id) = 65, code_id, "Not a valid code")
```

This expression for a computed field checks the case of the first character of lname and if it is lowercase makes it uppercase. Type the expression on a single line:

```
IF (Asc(lname) > 64 AND Asc(lname) < 91, lname,
WordCap(lname))
```

See also

Char
WordCap
Asc in Chapter 1, "PowerScript Functions"

Avg

Description

Calculates the average of the values of the column.

Syntax

Avg (*column* { for *range* { DISTINCT { *expres1* { , *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the average of the data values. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.

Parameter	Description
for *range* (optional)	If you specify *range*, you must precede it with the keyword for. Values for *range* are: ♦ All — (Default) The average value of all rows in *column*. ♦ Crosstab — (Crosstabs only) The average value of all rows in *column* in the crosstab. ♦ Graph — (Graphs only) The average value of all rows in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The average value of all rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The average value of the rows in *column* on a page.
DISTINCT (optional)	Causes Avg to consider only the distinct values in *column* when calculating the average. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The numeric data type of the column. Returns the average of the values of the rows in *range*.

Usage

If you specify *range*, Avg returns the average value of *column* in *range*. If you specify DISTINCT, Avg returns the average value of the distinct values in *column*, or if you specify *expresn*, the average of *column* for each distinct value of *expresn*.

In calculating the average, NULL values are ignored.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the average of the values in the column named salary:

```
Avg(salary)
```

This expression returns the average of the values in group 1 in the column named salary:

```
Avg(salary for group 1)
```

This expression returns the average of the values in column 5 on the current page:

```
Avg(#5 for page)
```

This computed field, typed on a single line, returns Above Average if the average salary for the page is greater than the average for the report:

```
If(Avg(salary for page) > Avg(salary), "Above
Average", " ")
```

This expression for a graph value sets the data to the average value of the sale_price column:

```
Avg(sale_price)
```

This expression for a graph value entered in the Graph Data window sets the data value to the average value of the sale_price column for the entire graph:

```
Avg(sale_price for graph)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the average of the order amount for the distinct order numbers:

```
Avg(order_amt for all DISTINCT order_nbr)
```

See also

Median
Mode

Bitmap

Description

Displays the specified bitmap in the DataWindow.

> **For computed fields only**
> You can use the Bitmap function *only* in a computed field.

Syntax

Bitmap (*string*)

Parameter	Description
string	A column containing bitmap files, a string containing the name of an image file (a BMP, RLE, or WMF file), or an expression that evaluates to a string containing the name of an image file

Return value

The special data type bitmap, which *cannot* be used in any other function.

Usage

Use Bitmap to dynamically display a bitmap in a computed field of a report or DataWindow. When *string* is a column containing bitmap files, the DataWindow can display a different bitmap on each row of the detail area.

Examples

The following examples are expressions for a computed field.

This expression dynamically displays the bitmap file contained in the column named employees:

```
Bitmap(employees)
```

If the employees column is column 3, the next expression gives the same result as the expression above:

```
Bitmap(#3)
```

The following expression displays the bitmap TOOLS.BMP:

```
Bitmap("TOOLS.BMP")
```

The following expression tests the value in the column named password and then uses the value to determine which bitmap to display:

```
Bitmap(If(password = "y", "yes.bmp", "no.bmp"))
```

Ceiling

Description

Determines the smallest whole number that is greater than or equal to a specified limit.

Syntax

Ceiling (*n*)

Parameter	Description
n	The number for which you want the smallest whole number that is greater than or equal to it

Return value

The data type of *n*. Returns the smallest whole number that is greater than or equal to *n*.

Examples

These expressions both return -4:

```
Ceiling(-4.2)
Ceiling(-4.8)
```

This expression for a computed field returns ERROR if the value in discount_amt is greater than the smallest whole number that is greater than or equal to discount_factor times price. Otherwise, it returns discount_amt. The expression is entered on a single line:

```
If(discount_amt <= Ceiling(discount_factor * price),
String(discount_amt), "ERROR")
```

To pass this validation rule, the value in discount_amt must be less than or equal to the smallest whole number that is greater than or equal to discount_factor times price:

```
discount_amt <= Ceiling(discount_factor * price)
```

See also

Int
Round
Truncate
Ceiling in Chapter 1, "PowerScript Functions"

Char

Description

Converts an integer to a character.

Syntax

Char (*n*)

Parameter	Description
n	The integer you want to convert to a character.

Return value

String. Returns the character whose ASCII value is *n*.

Example

This expression returns the escape character:

```
Char(27)
```

See also

Asc
Char in Chapter 1, "PowerScript Functions"

Cos

Description

Calculates the cosine of an angle.

Syntax

Cos (*n*)

Parameter	Description
n	The angle (in radians) for which you want the cosine

Return value

Double. Returns the cosine of *n*.

Examples

This expression returns 1:

```
Cos(0)
```

This expression returns .540302:

```
Cos(1)
```

This expression returns -1:

```
Cos(Pi(1))
```

See also

Pi
Sin
Tan
Cos in Chapter 1, "PowerScript Functions"

Count

Description

Calculates the total number of rows in the specified column.

Syntax

Count (*column* { for *range* { DISTINCT {*expres1*{, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the number of rows. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The total number of rows in *column*. ♦ Crosstab — (Crosstabs only) The total number of rows in *column* in the crosstab. ♦ Graph — (Graphs only) The total number of rows in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The number of rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The number of rows in *column* on a page.
DISTINCT (optional)	Causes Count to consider only the distinct values in *column* when counting the rows. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.

Parameter	Description
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

Long. Returns the total number of rows in *range*.

Usage

If you specify *range*, Count determines the number of rows in *column* in *range*. If you specify DISTINCT, Count returns the number of the distinct rows displayed in *column*, or if you specify *expresn*, the number of rows displayed in *column* where the value of *expresn* is distinct.

Null values in the column are ignored and are not included in the count.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the number of rows in the column named emp_id that are not NULL:

```
Count(emp_id)
```

This expression returns the number of rows in the column named emp_id of group 1 that are not NULL:

```
Count(emp_id for group 1)
```

This expression returns the number of dept_ids that are distinct:

```
Count(dept_id for all DISTINCT)
```

This expression returns the number of regions with distinct products:

```
Count(region_id for all DISTINCT Lower(product_id))
```

This expression returns the number of rows in column 3 on the page that are not NULL:

```
Count(#3 for page)
```

CrosstabAvg

Description

Calculates the average of the values returned by an expression in the values list of the crosstab.

> **For crosstabs only**
> You can use this function *only* in a crosstab report.

Syntax

CrosstabAvg (*n*)

Parameter	Description
n	The number of the crosstab-values expression for which you want the average of the returned values. The crosstab expression must be numeric.

Return value

Double. Returns the average of the crosstab values returned by expression *n*.

Usage

This function is meaningful *only* for the average of the values of the expression in a *row* in the crosstab.

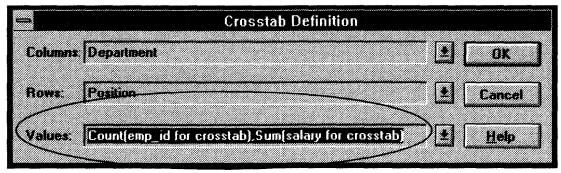
NULL values are ignored and are not included in the average.

> **Reviewing the expressions**
> To review the expressions defined for the crosstab values, open the Crosstab Definition window. To open the window, click the right mouse button in an unused area of the crosstab and then select Crosstab from the popup menu.

Examples

These examples use the crosstab expressions shown below:

```
Count(emp_id for crosstab),Sum(salary for crosstab)
```



This expression for a computed field in the crosstab returns the average of the employee counts (the first expression):

```
CrosstabAvg(1)
```

This expression for a computed field in the crosstab returns the average of the salary totals (the second expression):

```
CrosstabAvg(2)
```

See also

Avg
CrosstabCount
CrosstabMax
CrosstabMin
CrosstabSum

CrosstabCount

Description

Counts the number of values returned by an expression in the values list of the crosstab.

> **For crosstabs only**
> You can use this function *only* in a crosstab report.

Syntax

CrosstabCount (*n*)

Parameter	Description
n	The number of the crosstab-values expression for which you want the total number of returned values

Return value

Long. Returns the number of values returned by expression *n*.

Usage

This function is meaningful *only* for the count of the values of the expression in a *row* in the crosstab.

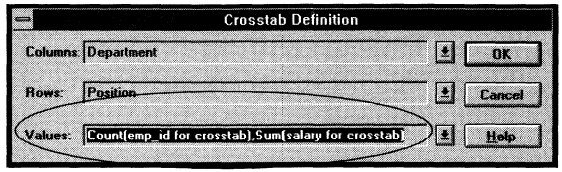
NULL values are ignored and are not included in the count.

> **Reviewing the expressions**
> To review the expressions defined for the crosstab values, open the Crosstab Definition window. To open the window, click the right mouse button in an unused area of the crosstab and then select Crosstab from the popup menu.

Examples

These examples use the crosstab-values expressions shown below:

```
Count(emp_id for crosstab),Sum(salary for crosstab)
```



This expression for a computed field in the crosstab returns the count of the employee counts (the first expression):

```
CrosstabCount(1)
```

This expression for a computed field in the crosstab returns the count of the salary totals (the second expression):

```
CrosstabCount(2)
```

See also

Count
CrosstabAvg
CrosstabMax
CrosstabMin
CrosstabSum

CrosstabMax

Description

Calculates the maximum value returned by an expression in the values list of the crosstab.

> **For crosstabs only**
> You can use this function *only* in a crosstab report.

Syntax

CrosstabMax (*n*)

Parameter	Description
n	The number of the crosstab-values expression for which you want the maximum returned value. The expression's data type must be numeric.

Return value

Double. Returns the maximum value returned by expression *n*.

Usage

This function is meaningful *only* for the maximum of the values of the expression in a *row* in the crosstab.

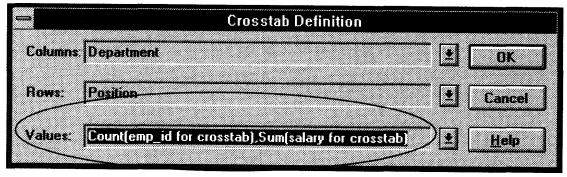
NULL values are ignored and are not included in the comparison.

> **Reviewing the expressions**
> To review the expressions defined for the crosstab values, open the Crosstab Definition window. To open the window, click the right mouse button in an unused area of the crosstab and then select Crosstab from the popup menu.

Examples

These examples use the crosstab-values expressions shown below:

```
Count(emp_id for crosstab),Sum(salary for crosstab)
```



This expression for a computed field in the crosstab returns the maximum of the employee counts (the first expression):

```
CrosstabMax(1)
```

This expression for a computed field in the crosstab returns the maximum of the salary totals (the second expression):

```
CrosstabMax(2)
```

See also

CrosstabAvg
CrosstabCount
CrosstabMin

CrosstabSum
Max

CrosstabMin

Description

Calculates the minimum value returned by an expression in the values list of the crosstab.

> **For crosstabs only**
> You can use this function *only* in a crosstab report.

Syntax

CrosstabMin (*n*)

Parameter	Description
n	The number of the crosstab-values expression for which you want the minimum return value. The expression's data type must be numeric.

Return value

Double. Returns the minimum value returned by expression *n*.

Usage

This function is meaningful *only* for the minimum of the values of the expression in a *row* in the crosstab.

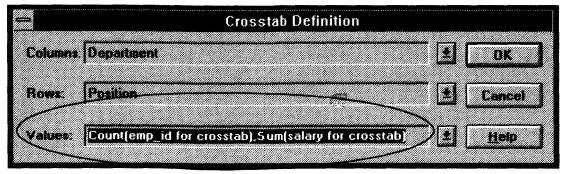
NULL values are ignored and are not included in the comparison.

> **Reviewing the expressions**
> To review the expressions defined for the crosstab values, open the Crosstab Definition window. To open the window, click the right mouse button in an unused area of the crosstab and then select Crosstab from the popup menu.

Examples

These examples use the crosstab-values expressions shown below:

```
Count(emp_id for crosstab),Sum(salary for crosstab)
```



This expression for a computed field in the crosstab returns the minimum of the employee counts (the first expression):

```
CrosstabMin(1)
```

This expression for a computed field in the crosstab returns the minimum of the salary totals (the second expression):

```
CrosstabMin(2)
```

See also

CrosstabAvg
CrosstabCount
CrosstabMax
CrosstabSum
Min

CrosstabSum

Description

Calculates the sum of the values returned by an expression in the values list of the crosstab.

> **For crosstabs only**
> You can use this function *only* in a crosstab report.

Syntax

CrosstabSum (*n*)

Parameter	Description
n	The number of the crosstab-values expression for which you want the sum of the returned values. The expression's data type must be numeric.

Return value

Double. Returns the total of the values returned by expression *n*.

Usage

This function is meaningful *only* for the sum of the values of the expression in a *row* in the crosstab.

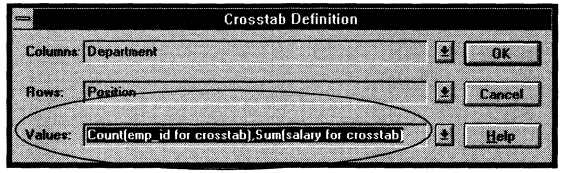
NULL values are ignored and are not included in the sum.

> **Reviewing the expressions**
> To review the expressions defined for the crosstab values, open the Crosstab Definition window. To open the window, click the right mouse button in an unused area of the crosstab and then select Crosstab from the popup menu.

Examples

These examples use the crosstab-values expressions shown below:

```
Count(emp_id for crosstab),Sum(salary for crosstab)
```



This expression for a computed field in the crosstab returns the sum of the employee counts (the first expression):

```
CrosstabSum(1)
```

This expression for a computed field in the crosstab returns the sum of the salary totals (the second expression):

```
CrosstabSum(2)
```

See also

CrosstabAvg
CrosstabCount
CrosstabMax
CrosstabMin
Sum

CumulativePercent

Description

Calculates the total value of the rows up to and including the current row in the specified column as a percentage of the total value of the column (a running percentage).

Syntax

CumulativePercent (*column* { for *range* })

Parameter	Description
column	The column for which you want the cumulative value of the rows up to and including the current row as a percentage of the total value of the column for *range*. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The cumulative percentage of all rows in *column*. ♦ Crosstab — (Crosstabs only) The cumulative percentage of all rows in *column* in the crosstab. ♦ Graph — (Graphs only) The cumulative percentage of all rows in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The cumulative percentage of all rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The cumulative percentage of all rows in *column* on a page.

Return value

Long. Returns the cumulative percentage value.

Usage

If you specify *range*, CumulativePercent restarts the accumulation at the start of the range.

In calculating the percentage, NULL values are ignored.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the running percentage for the values that are not NULL in the column named salary:

```
CumulativePercent(salary)
```

This expression returns the running percentage for the column named salary for the values in group 1 that are not NULL:

```
CumulativePercent(salary for group 1)
```

This expression entered in the Values box in the Graph Data window returns the running percentage for the salary column for the values in the graph that are not NULL:

```
CumulativePercent(salary for graph)
```

This expression in a crosstab computed field returns the running percentage for the salary column for the values in the crosstab that are not NULL:

```
CumulativePercent(salary for crosstab)
```

See also

Percent
CumulativeSum

CumulativeSum

Description

Calculates the total value of the rows up to and including the current row in the specified column (a running total).

Syntax

CumulativeSum (*column* { for *range* })

Parameter	Description
column	The column for which you want the cumulative total value of the rows up to and including the current row for group. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.

Parameter	Description
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The cumulative sum of all rows in *column*. ♦ Crosstab — (Crosstabs only) The cumulative sum of all rows in *column* in the crosstab. ♦ Graph — (Graphs only) The cumulative sum of all rows in *column* for the graph. This value for range has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The cumulative sum of all rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The cumulative sum of all rows in *column* on a page.

Return value

Long. Returns the cumulative total value of the rows.

Usage

If you specify *range*, CumulativeSum restarts the accumulation at the start of the range.

In calculating the sum, NULL values are ignored.

Examples

This expression returns the running total for the values that are not NULL in the column named salary:

```
CumulativeSum(salary)
```

This expression returns the running total for the values that are not NULL in the column named salary in group 1:

```
CumulativeSum(salary for group 1)
```

This expression returns the running total for the values that are not NULL in the column named salary in group 1:

```
CumulativeSum(salary for group 1)
```

This expression entered in the Values box in the Graph Data window returns the running total for the salary column for the values in the graph that are not NULL:

```
CumulativeSum(salary for graph)
```

This expression in a crosstab computed field returns the running total for the salary column for the values in the crosstab that are not NULL:

```
CumulativeSum(salary for crosstab)
```

See also CumulativePercent

Date

Description Converts a string whose value is a valid date to a value of data type date.

Syntax **Date** (*string*)

Parameter	Description
string	A string containing a valid date (such as Jan 1, 1998, or 12-31-99) that you want returned as a date

Return value Date. Returns the date in *string* as a date. If *string* does not contain a valid date, Date returns NULL.

Usage The value of the string must be a valid date.

> **Valid dates**
> Valid dates can include any combination of day (1–31), month (1–12 or the name or abbreviation of a month), and year (two or four digits). Leading zeros are optional for month and day. If the month is a name or an abbreviation, it can come before or after the day; if it is a number, it must be in the month location specified in the Windows control panel. A four-digit number is assumed to be a year.
>
> If the year is two digits, then PowerBuilder chooses the century, as follows. If the year is between 00 and 49, PowerBuilder assumes 20 as the first two digits; if it is between 50 and 99, PowerBuilder assumes 19. If your data includes date before 1950, such as birth dates, always specify a four-digit year so that PowerBuilder interprets the date as intended.
>
> PowerBuilder handles years from 1000 to 3000 inclusive.

A DataWindow expression has a more limited set of data types than the functions that can be part of the expression. Although the Date function returns a date value, the whole expression in the DataWindow is promoted to a DateTime value. Therefore, if your expression consists of a single Date function, it will appear that Date returns the wrong data type. To display the date without the time, choose an appropriate display format. (See "Using DataWindow Painter Functions" at the beginning of this chapter.)

Examples

These expressions all return the date data type for July 4, 1994 (1994-07-04) when the default position of month is center:

```
Date("1994/07/04")
Date("1994 July 4")
Date("July 4, 1994")
```

See also

IsDate
Date in Chapter 1, "PowerScript Functions"

DateTime

Description

Combines a date and a time value into a DateTime value.

Syntax

DateTime (*date* {, *time* })

Parameter	Description
date	A valid date (such as Jan 1, 1998, or 12-31-99) or a blob variable whose first value is a date that you want included in the value returned by DateTime.
time (optional)	A valid time (such as 8AM or 10:25:23:456799) or a blob variable whose first value is a time that you want included in the value returned by DateTime. If you include a time, only the hour portion is required. If you omit the minutes, seconds, or microseconds, they are assumed to be 0s. If you omit AM or PM, the hour is determined according to the 24-hour clock.

Return value

DateTime. Returns a DateTime value based on the values in *date* and optionally *time*. If time is omitted, DateTime uses 00:00:00.000000 (midnight).

Usage

For information on valid dates, see Date.

To display microseconds in a time, the display format for the field must include microseconds.

Example

This expression returns the values in the order_date and order_time columns as a DateTime value that can be used to update the database:

```
DateTime(Order_Date, Order_Time)
```

See also

Date
Time
DateTime in Chapter 1, "PowerScript Functions"

Day

Description

Obtains the day of the month in a date value.

Syntax

Day (*date*)

Parameter	Description
date	The date from which you want the day

Return value

Integer. Returns an integer (1–31) representing the day of the month in *date*.

Example

This expression returns 31:

```
Day(1994-01-31)
```

This expression returns the day of the month in the start_date column:

```
Day(start_date)
```

See also

Date
IsDate
Month
Year
Day in Chapter 1, "PowerScript Functions"

DayName

Description

Determines the day of the week in a date value and returns the weekday's name.

Syntax

DayName (*date*)

Parameter	Description
date	The date for which you want the name of the day

Return value

String. Returns a string whose value is the name of the weekday (Sunday, Monday, and so on) for *date*.

Example

This expression for a computed field returns Okay if the day in date_signed is not Sunday. The expression is entered on a single line:

```
If (DayName(date_signed) <> "Sunday", "Okay",
"Invalid Date")
```

To pass this validation rule, the day in date_signed must not be Sunday:

```
DayName(date_signed) <> "Sunday"
```

See also

Date
Day
DayNumber
IsDate
DayName in Chapter 1, "PowerScript Functions"

DayNumber

Description

Determines the day of the week of a date value and returns the number of the weekday.

Syntax

DayNumber (*date*)

Parameter	Description
date	The date from which you want the number of the day of the week

Return value

Integer. Returns an integer (1–7) representing the day of the week of *date*. Sunday is day 1, Monday is day 2, and so on.

Examples

This expression for a computed field returns Wrong Day if the date in start_date is not a Sunday or a Monday:

```
If(DayNumber(start_date) > 2, "Okay", "Wrong Day")
```

This expression for a computed field returns Wrong Day if the date in end_date is a Saturday or a Sunday. The expression is entered on a single line:

```
If(DayNumber(end_date) > 1 and DayNumber(end_date) <
7, "Okay", "Wrong Day")
```

This validation rule for the column end_date ensures that the day is not a Saturday or Sunday:

```
DayNumber(end_date) >1 and DayNumber(end_date) < 7
```

See also

Date
Day
DayName
IsDate
DayNumber in Chapter 1, "PowerScript Functions"

DaysAfter

Description

Determines the number of days one date occurs after another.

Syntax

DaysAfter (*date1*, *date2*)

Parameter	Description
date1	A date value that is the start date of the interval being measured
date2	A date value that is the end date of the interval

Return value

Long. Returns a long containing the number of days *date2* occurs after *date1*. If *date2* occurs before *date1*, DaysAfter returns a negative number.

Examples

This expression returns 4:

```
DaysAfter(1995-12-20, 1995-12-24)
```

This expression returns -4:

```
DaysAfter(1995-12-24, 1995-12-20)
```

This expression returns 0:

```
DaysAfter(1995-12-24, 1995-12-24)
```

This expression returns 5:

```
DaysAfter(1994-12-29, 1995-01-03)
```

See also

Date
SecondsAfter
DaysAfter in Chapter 1, "PowerScript Functions"

Describe

Description

Reports the values of attributes of a DataWindow object and objects within the DataWindow object. Each column and graphic object in the DataWindow has a set of attributes, which are listed in Appendix A. You specify one or more attributes as a string and Describe returns the values of the attributes.

Syntax

Describe (*attributelist*)

Parameter	Description
attributelist	A string whose value is a blank-separated list of attributes or Evaluate functions. For a list of valid attributes, see Appendix A, "DataWindow Object Attributes."

Return value

String. Returns a string that includes a value for each attribute or Evaluate function. A newline character (~n) separates the value of each item in *attributelist*.

If the attribute list contains an invalid item, Describe returns an exclamation point (!) for that item and ignores the rest of the attribute list. Describe returns a question mark (?) if there is no value for an attribute.

Usage

The values for *attributelist* can be quite complex. For complete information and examples, see Describe in Chapter 1.

Example

This expression for a computed field in the DataWindow's header band displays the DataWindow object's SELECT statement:

```
Describe("DataWindow.Table.Select")
```

See also

Describe in Chapter 1, "PowerScript Functions"

Exp

Description

Raises *e* to the specified power.

Syntax

Exp (*n*)

Parameter	Description
n	The power to which you want to raise *e* (2.71828)

Return value

Double. Returns *e* raised to the power *n*.

Example

This expression returns 7.38905609893065:

```
Exp(2)
```

See also

Log
LogTen
Exp in Chapter 1, "PowerScript Functions"

Fact

Description

Determines the factorial of a number.

Syntax

Fact (*n*)

Parameter	Description
n	The number for which you want the factorial

Return value

Double. Returns the factorial of *n*.

Examples

This expression returns 24:

```
Fact(4)
```

Both these expressions return 1:

```
Fact(1)
```

```
Fact(0)
```

See also

Fact in Chapter 1, "PowerScript Functions"

Fill

Description

Builds a string of the specified length by repeating the specified characters until the result string is long enough.

Syntax

Fill (*chars, n*)

Parameter	Description
chars	A string whose value will be repeated to fill the return string
n	A long whose value is the length of the string you want returned

Return value

String. Returns a string *n* characters long filled with repetitions of the characters in the argument *chars*. If the argument *chars* has more than *n* characters, the first *n* characters of *chars* are used to fill the return string. If the argument *chars* has fewer than *n* characters, the characters in *chars* are repeated until the return string has *n* characters.

Usage

Fill is used to create a line or other special effect. For example, asterisks repeated in a printed report can fill an amount line, or hyphens can simulate a total line in a screen display.

Examples

This expression returns a string containing 35 stars:

```
Fill("*", 35)
```

This expression returns the string -+-+-+-:

```
Fill("-+", 7)
```

This expression returns 10 tildes (~):

```
Fill("~", 10)
```

See also

Space
Fill in Chapter 1, "PowerScript Functions"

First

Description

Determines the value in the first row in the specified column.

Syntax

First (*column* {for *range* { DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the value of the first row. *Column* can be a column name or a column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The value of the first row in *column*. ♦ Crosstab — (Crosstabs only) The value of the first row in *column* in the crosstab. ♦ Graph — (Graphs only) The value of the first row in *column* for the graph. This value for *range* for *group* has effect only when you specify Page in the Rows in the Graph Data window. ♦ GroupNbr — The value of the first row in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The value of the first row in *column* on a page.
DISTINCT (optional)	Causes First to consider only the distinct values in *column* when determining the first value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of the column. Returns the value in the first row of *column*. If you specify *range*, First returns the value of the first row in *column* in *range*.

Usage

If you specify *range*, First determines the value of the first row in *column* in *range*. If you specify DISTINCT, First returns the first distinct value in *column*, or if you specify *expresn*, the first distinct value in *column* where the value of *expresn* is distinct.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the first value in column 3 on the page:

```
First(#3 for page)
```

This expression returns the first distinct value in the column named dept_id in group 2:

```
First(dept_id for group 2 DISTINCT)
```

This expression returns the first distinct value in the column named dept_id in group 2:

```
First(dept_id for group 2 DISTINCT)
```

This expression returns the first value in the column named dept_id in group 2:

```
First(dept_id for group 2)
```

See also

Last

GetRow

Description

Reports the number of a row associated with a band in a DataWindow control.

Syntax

GetRow ( )

Return value

Long. Returns the number of a row if it succeeds, *0* if no data has been retrieved or added, and *-1* if an error occurs. Where you call GetRow determines what row it returns, as follows.

If the object in the DataWindow is in this band	GetRow returns
Header	First row on the page
Group header	First row in the group
Detail	The row in which the expression occurs
Group trailer	Last row in the group
Summary	Last row in the report or DataWindow
Footer	Last row on the page

Example

This expression for a computed field in the detail band displays the number of each row:

```
GetRow( )
```

This expression for a computed field in the header band checks to see if there is data; it returns the number of the first row on the page if there is data, and otherwise returns No Data:

```
If(GetRow( ) = 0, "No Data", String(GetRow( )))
```

See also

GetRow in Chapter 1, "PowerScript Functions"

GetText

Description

Obtains the text that the user has entered in a column.

Syntax

GetText ( )

Return value

String. Returns the text the user has entered in the current column.

Usage

Use GetText in validation rules to compare what the user has entered to application-defined criteria before it is accepted into the data buffer.

Examples

This validation rule checks that the value the user entered in the column is less than 100:

```
Integer(GetText( )) < 100
```

See also

GetText in Chapter 1, "PowerScript Functions"

Hour

Description

Obtains the hour in a time value. The hour is based on a 24-hour clock.

Syntax

Hour (*time*)

Parameter	Description
time	The time value from which you want the hour

Return value

Integer. Returns an integer (00–23) containing the hour portion of *time*.

Examples

This expression returns the current hour:

```
Hour(Now( ))
```

This expression returns 19:

```
Hour(19:01:31)
```

See also

Minute
Now
Second
Hour in Chapter 1, "PowerScript Functions"

If

Description

Evaluates a condition and returns a value based on that condition.

Syntax

If (*boolean, truevalue, falsevalue*)

Parameter	Description
boolean	A boolean expression that evaluates to TRUE or FALSE
truevalue	A string containing the value you want returned if the boolean expression is TRUE
falsevalue	A string containing the value you want returned if the boolean expression is FALSE

Return value

The data type of *truevalue* or *falsevalue*. Returns *truevalue* if *boolean* is TRUE and *falsevalue* if it is FALSE. Returns NULL if an error occurs.

Examples

This expression returns Boss if salary is over $100,000 and Employee if salary is less than or equal to $100,000:

```
If(salary > 100000, "Boss", "Employee")
```

The following expression returns Boss if salary is over $100,000, Supervisor if salary is between $12,000 and $100,000, and Clerk if salary is less than or equal to $12,000. Note that you enter the expression as a single line:

```
If(salary > 100000, "Boss", If(salary > 12000,
"Supervisor", "Clerk"))
```

In this example of a validation rule, the value the user should enter in the commission column depends on the price. If price is greater than or equal to 1000, then the commission is between .10 and .20. If price is less than 1000, then the commission must be between .04 and .09. The validation rule is:

```
(Number(GetText()) >= If(price >=1000, .10, .04))
AND (Number(GetText()) <= If(price >=1000, .20,
.09))
```

The accompanying error message expression might be:

```
"Price is " +  If(price >= 1000, "greater than or
equal to", "less than") + " 1000. Commission must be
between " + If(price >= 1000, ".10", ".04") +  " and
" + If(price >= 1000, ".20.", ".09.")
```

Int

Description

Determines the largest whole number less than or equal to a number.

Syntax

Int (*n*)

Parameter	Description
n	The number for which you want the largest whole number that is less than or equal to it

Return value

The data type of *n*. Returns the largest whole number less than or equal to *n*.

Examples

These expressions return 3.0:

```
Int(3.2)
Int(3.8)
```

These expressions return -4.0:

```
Int(-3.2)
Int(-3.8)
```

See also

Ceiling
Integer
Round
Truncate
Int in Chapter 1, "PowerScript Functions"

Integer

Description

Converts the value of a string to an integer.

Syntax

Integer (*string*)

Parameter	Description
string	The string you want returned as an integer

Return value

Integer. Returns the contents of *string* as an integer if it succeeds and *0* if *string* is not a number.

Examples

This expression converts the string 24 to an integer:

```
Integer("24")
```

This expression for a computed field returns Not a valid age if age does not contain a number. The expression checks whether the Integer function returns 0, which means it failed to convert the value:

```
If (Integer(age) <> 0, age, "Not a valid age")
```

This expression returns 0:

```
Integer("3ABC")   // 3ABC is not a number
```

This validation rule checks that the value in the column the user entered is less than 100:

```
Integer(GetText()) < 100
```

This validation rule for the column named age insures that age contains a string:

```
Integer(age) <> 0
```

See also

IsNumber
Integer in Chapter 1, "PowerScript Functions"

IsDate

Description

Tests whether a string value is a valid date.

Syntax

IsDate (*datevalue*)

Parameter	Description
datevalue	A string whose value you want to test to determine whether it is a valid date

Return value

Boolean. Returns TRUE if *datevalue* is a valid date and FALSE if it is not.

Examples

This expression returns TRUE:

```
IsDate("Jan 1, 95")
```

This expression returns FALSE:

```
IsDate("Jan 32, 1997")
```

This expression for a computed field is entered on a single line in the Computed Field Definition Window. It returns the number of the day in date_received in the computed field if the column contains a valid date, otherwise it returns 0:

```
If(IsDate(String(date_received)),
DayNumber(date_received), 0)
```

See also

IsDate in Chapter 1, "PowerScript Functions"

IsNull

Description

Reports whether the value of a column or expression is NULL.

Syntax

IsNull (*any*)

Parameter	Description
any	A column or expression that you want to test to determine whether its value is NULL

Return value

Boolean. Returns TRUE if *any* is NULL and FALSE if it is not.

Usage

Use IsNull to test whether a user-entered value or a value retrieved from the database is NULL.

Examples

This expression returns TRUE if either a or b is NULL:

```
IsNull(a + b)
```

This expression returns TRUE if the value in the salary column is NULL:

```
IsNull(salary)
```

This expression returns TRUE if the value the user has entered is NULL:

```
IsNull(GetText( ))
```

See also

IsNull in Chapter 1, "PowerScript Functions"

IsNumber

Description

Reports whether the value of a string is a number.

Syntax

IsNumber (*string*)

Parameter	Description
string	A string whose value you want to test to determine whether it is a valid PowerScript number

Return value

Boolean. Returns TRUE if *string* is a valid PowerScript number and FALSE if it is not.

Examples

This expression returns TRUE:

```
IsNumber("32.65")
```

This expression returns FALSE:

```
IsNumber("A16")
```

This expression for a computed field returns Not a valid age if age does not contain a number:

```
If (IsNumber(age), age, "Not a valid age")
```

To pass this validation rule, Age_nbr must be a number:

```
IsNumber(Age_nbr) = TRUE
```

See also

Integer
IsNumber in Chapter 1, "PowerScript Functions"

IsRowModified

Description

Reports whether the row has been modified.

Syntax

IsRowModified ( )

Return value

Boolean. Returns TRUE if the row has been modified and FALSE if it has not.

Usage

When you call IsRowModified in bands other than the detail band, it reports on a row in the detail band. See GetRow for a table specifying which row is associated with each band for reporting purposes.

Examples

This expression in a computed field in the detail area displays TRUE or FALSE to indicate whether each row has been modified:

```
IsRowModified( )
```

This expression for the Color attribute in the Attribute Expressions dialog displays the associated column text in red if the user has modified any value in the row:

```
If(IsRowModified( ), 255, 0)
```

See also

GetRow

IsRowNew

Description

Reports whether the row has been newly inserted in the DataWindow.

Syntax

IsRowNew ( )

Return value

Boolean. Returns TRUE if the row is new and FALSE if it was retrieved from the database.

Usage

When you call IsRowNew in bands other than the detail band, it reports on a row in the detail band. See GetRow for a table specifying which row is associated with each band for reporting purposes.

Examples

This expression for the Protect attribute in the Attribute Conditional Expressions dialog prevents the user from modifying the associated column unless the row has been newly inserted:

```
If(IsRowNew( ), 0, 1)
```

See also

GetRow
GetItemStatus in Chapter 1, "PowerScript Functions"

IsSelected

Description

Determines whether the row is selected. A selected row is highlighted using reverse video.

Syntax

IsSelected ( )

Return value

Boolean. Returns TRUE if the row is selected and FALSE if it is not selected.

Usage

When you call IsSelected in bands other than the detail band, it reports on a row in the detail band. See GetRow for a table specifying which row is associated with each band for reporting purposes.

Example

This expression for a computed field in the detail area displays a bitmap if the row is selected:

```
Bitmap(If(IsSelected( ), "beach.bmp", ""))
```

This example allows the DataWindow to display a salary total for all the selected rows. The expression for a computed field in the detail band returns the salary only when the row is selected so that another computed field in the summary band can add up all the selected salaries.

The expression for cf_selected_salary, the computed field in the detail band, is:

```
If(IsSelected( ), salary, 0)
```

The expression for the computed field in the summary band is:

```
Sum(cf_selected_salary for all)
```

See also

GetRow
IsSelected in Chapter 1, "PowerScript Functions"

IsTime

Description

Reports whether the value of a string is a valid time value.

Syntax

IsTime (*timevalue*)

Parameter	Description
timevalue	A string whose value you want to test to determine whether it is a valid time

Return value

Boolean. Returns TRUE if *timevalue* is a valid time and FALSE if it is not.

Examples

This expression returns TRUE:

```
IsTime("8:00:00 am")
```

This expression returns FALSE:

```
IsTime("25:00")
```

To pass this validation rule, the value in start_time must be a time:

```
IsTime(start_time)
```

See also

IsTime in Chapter 1, "PowerScript Functions"

Large

Description

Finds a large value at a specified ranking in a column (for example, third largest, fifth largest) and returns the value of another column or expression based on the result.

Syntax

Large (*returnexp*, *column*, *ntop* { FOR *range* { DISTINCT { *expres1* {, *expres2* {, ...}}}}})

Parameter	Description
returnexp	The value you want returned when the large value is found. *Returnexp* includes a reference to a column, but not necessarily the column that is being evaluated for the largest value, so that a value is returned from the same row that contains the large value.
column	The column that contains the large value you are searching for. *Column* can be a column name or a column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
ntop	The ranking of the large value in relation to the column's largest value. For example, when *ntop* is 2, Large finds the second largest value.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The large value in *column*. ♦ Crosstab — (Crosstabs only) The large value in *column* in the crosstab. ♦ Graph — (Graphs only) The large value in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows in the Graph Data window. ♦ GroupNbr — The large value in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The large value in *column* on a page.
DISTINCT (optional)	Causes Large to consider only the distinct values in *column* when determining the large value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.

Parameter	Description
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of *returnexp*. Returns the *ntop* largest value if it succeeds and *-1* if an error occurs.

Usage

If you specify *range*, Large returns the value in *returnexp* when the value in *column* is the *ntop* largest value in *range*. If you specify DISTINCT, Large returns *returnexp* when the value in *column* is the *ntop* largest value of the distinct values in *column*, or if you specify *expresn*, the *ntop* largest for each distinct value of *expresn*.

> **Tip**
> If you don't need a return value from another column and you want to find the largest value (*ntop* = 1), use Max; it is faster.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Example

These expressions return the names of the salespeople with the three largest sales (sum_sales is the sum of the sales for each salesperson) in group 2, which might be the salesregion group. Note that sum_sales contains the values being compared, but Large returns a value in the name column:

```
Large(name, sum_sales, 1 for group 2)
Large(name, sum_sales, 2 for group 2)
Large(name, sum_sales, 3 for group 2)
```

This example reports the salesperson with the third largest sales, considering only the first entry for each person:

```
Large(name, sum_sales, 3 for all DISTINCT sum_sales)
```

See also

Small

Last

Description

Determines the value in the last row in the specified column.

Syntax

Last (*column* {for *range* {, DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the value of the last row. *Column* can be a column name or a column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The value of the last row in *column*. ♦ Crosstab — (Crosstabs only) The value of the last row in *column* in the crosstab. ♦ Graph — (Graphs only) The value of the last row in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The value of the last row in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The value of the last row in *column* on a page.
DISTINCT (optional)	Causes Last to consider only the distinct values in *column* when determining the last value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of the column. Returns the value in the last row of *column*. If you specify *range*, Last returns the value of the last row in *column* in *range*.

Usage

If you specify *range*, Last determines the value of the last row in *column* in *range*. If you specify DISTINCT, Last returns the last distinct value in *column*, or if you specify *expresn*, the last distinct value in *column* where the value of *expresn* is distinct.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the last distinct value in the column named dept_id in group 2:

```
Last(dept_id for group 2 DISTINCT)
```

This expression returns the last value in the column named emp_id in group 2:

```
Last(emp_id for group 2)
```

See also

First

Left

Description

Obtains a specified number of characters from the beginning of a string.

Syntax

Left (*string, n*)

Parameter	Description
string	The string containing the characters you want
n	A long specifying the number of characters you want

Return value

String. Returns the leftmost *n* characters in *string* if it succeeds and the empty string ("") if an error occurs.

If *n* is greater than or equal to the length of the string, Left returns the entire string. It does not add spaces to make the return value's length equal to *n*.

Examples

This expression returns BABE:

```
Left("BABE RUTH", 4)
```

This expression returns BABE RUTH:

```
Left("BABE RUTH", 40)
```

This expression for a computed field returns only the first 40 characters of the text in the column home_address:

```
Left(home_address, 40)
```

See also

Mid
Pos
Right
Left in Chapter 1, "PowerScript Functions"

LeftTrim

Description

Removes spaces from the beginning of a string.

Syntax

LeftTrim (*string*)

Parameter	Description
string	The string you want returned with leading spaces deleted

Return value

String. Returns a copy of *string* with leading spaces deleted if it succeeds and the empty string ("") if an error occurs.

Examples

This expression returns RUTH:

```
LeftTrim("   RUTH")
```

This expression for a computed field deletes any leading blanks from the value in the column lname and returns the value preceded by the salutation specified in salut_emp:

```
salut_emp + " " + LeftTrim(lname)
```

See also

RightTrim
Trim
LeftTrim in Chapter 1, "PowerScript Functions"

Len

Description

Reports the length of a string.

Syntax

Len (*string*)

Parameter	Description
string	The string for which you want the length

Return value

Long. Returns a long containing the length of *string* if it succeeds and *-1* if an error occurs.

Example

This expression returns 0:

```
Len("")
```

This expression for a computed field returns only the first 40 characters of home_address if the length is more than 40 characters:

```
Len(home_address, 40)
```

This validation rule tests that the value the user entered is less than 20 characters:

```
Len(GetText()) < 20
```

See also

Len in Chapter 1, "PowerScript Functions"

Log

Description

Determines the natural logarithm of a number.

Syntax

Log (*n*)

Parameter	Description
n	The number for which you want the natural logarithm (base e). The value of *n* must be greater than 0.

Return value

Double. Returns the natural logarithm of *n*. An execution error occurs if *n* is negative or zero.

> **Tip**
> The inverse of the Log function is the Exp function.

Examples

This expression returns 2.302585092:

```
Log(10)
```

This expression returns –.693147 . . .:

```
Log(0.5)
```

Both these expressions result in an error during execution:

```
Log(0)
```

```
Log(-2)
```

See also

Exp
LogTen
Log in Chapter 1, "PowerScript Functions"

LogTen

Description

Determines the base 10 logarithm of a number.

Syntax

LogTen (*n*)

Parameter	Description
n	The number for which you want the base 10 logarithm. The value of *n* must not be negative.

Return value

Double. Returns the decimal log of *n*.

> **Obtaining a number**
> The expression 10^n is the inverse for LogTen(*n*). To obtain *n* given number (nbr = LogTen(n)), use: *n* = 10^r.

Examples

This expression returns 1:

```
LogTen(10)
```

The following expressions both return 0:

```
LogTen(1)
LogTen(0)
```

This expression results in an execution error:

```
LogTen(-2)
```

See also

Log
LogTen in Chapter 1, "PowerScript Functions"

Long

Description

Converts the value of a string to a long.

Syntax

Long (*string*)

Parameter	Description
string	The string you want returned as a long

Return value

Long. Returns the contents of *string* as a long if it succeeds and *0* if *string* is not a valid number.

Example

This expression returns 2167899876 as a long:

```
Long("2167899876")
```

See also

Long in Chapter 1, "PowerScript Functions"

LookUpDisplay

Description

Obtains the display value in the code table associated with the data value in the specified column.

Syntax

LookUpDisplay (*column*)

Parameter	Description
column	The column for which you want the code table display value

Return value

String. Returns the display value when it succeeds and the empty string ("") if an error occurs.

Usage

If a column has a code table, the DataWindow's buffer stores a value from the data column of the code table, but the user sees a value from the display column. Use LookUpDisplay to get the value the user sees.

Code tables and data values and graphs
When a column that is displayed in a graph has a code table, the graph displays the data values of the code table by default. To display the display values, call this function when you define the graph data.

Examples

This expression returns the display value for the column unit_measure:

```
LookUpDisplay(unit_measure)
```

Assume the column product_type has a code table and you want to use it as a category for a graph. To display the product type descriptions instead of the data values in the categories, enter this expression in the Category box in the Graph Data window:

```
LookUpDisplay(product_type)
```

Lower

Description

Converts all the characters in a string to lowercase.

Syntax

Lower (*string*)

Parameter	Description
string	The string you want to convert to lowercase letters

Return value

String. Returns *string* with uppercase letters changed to lowercase if it succeeds and the empty string ("") if an error occurs.

Example

This expression returns babe ruth:

```
Lower("Babe Ruth")
```

See also

Upper
Lower in Chapter 1, "PowerScript Functions"

Match

Description

Determines whether a string's value contains a particular pattern of characters.

Syntax

Match (*string, textpattern*)

Parameter	Description
string	The string in which you want to look for a pattern of characters
textpattern	A string whose value is the text pattern

Return value

Boolean. Returns TRUE if *string* matches *textpattern* and FALSE if it does not. Match also returns FALSE if either argument has not been assigned a value or the pattern is invalid.

Usage

Match enables you to evaluate whether a string contains a general pattern of characters. To find out whether a string contains a specific substring, use the Pos function.

Textpattern is similar to a regular expression. It consists of metacharacters, which have special meaning, and ordinary characters, which match themselves. You can specify that the string begin or end with one or more characters from a set, or that it contain any characters except those in a set. See Chapter 3 for a list of metacharacters and their meaning, as well as sample patterns.

Example

This validation rule checks that the value the user entered begins with an uppercase letter. If value of the expression is false, the data fails validation:

```
Match(GetText( ), "^[A-Z]")
```

See also

Match in Chapter 1, "PowerScript Functions"
Chapter 3, "Using Text Patterns"

Max

Description

Determines the maximum value in the specified column.

Syntax

Max (*column* {for *range* { DISTINCT { *expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the maximum value. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The maximum value of *column*. ♦ Crosstab — (Crosstabs only) The maximum value of *column* in the crosstab. ♦ Graph — (Graphs only) The maximum value of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The maximum value of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The maximum value of *column* on a page.
DISTINCT (optional)	Causes Max to consider only the distinct values in *column* when determining the largest value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of the column. Returns the maximum value in the rows of *column*. If you specify *range*, Max returns the maximum value in *column* in *range*.

Usage

If you specify *range*, Max determines the maximum value in *column* in *range*. If you specify DISTINCT, Max returns the maximum distinct value in *column*, or if you specify *expresn*, the maximum distinct value in *column* where the value of *expresn* is distinct.

NULL values are ignored and are not considered in determining the maximum.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the maximum of the values in the age column on the page:

```
Max(age for page)
```

This expression returns the maximum of the values in column 3 on the page:

```
Max(#3 for page)
```

This expression returns the maximum of the values in the column named age in group 1:

```
Max(age for group 1)
```

Assume the report or DataWindow displays the order number, amount, and line items for each order. This computed field returns the maximum of the order amount for the distinct order numbers:

```
Max(order_amt for all DISTINCT order_nbr)
```

See also

Min
Max in Chapter 1, "PowerScript Functions"

Median

Description

Calculates the median of the values of the column. The median is the middle value in the set of values, for which there is an equal number of values greater and smaller than it.

Syntax

Median (*column* { for *range* { DISTINCT { *expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the median of the data values. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The median value of all rows in *column*. ♦ Crosstab — (Crosstabs only) The median value of all rows in *column* in the crosstab. ♦ Graph — (Graphs only) The median value of all rows in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The median value of all rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The median value of the rows in *column* on a page.
DISTINCT (optional)	Causes Median to consider only the distinct values in *column* when determining the median. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The numeric data type of the column. Returns the median of the values of the rows in *range* if it succeeds and *-1* if an error occurs.

Usage

If you specify *range*, Median returns the median value of *column* in *range*. If you specify DISTINCT, Median returns the median value of the distinct values in *column*, or if you specify *expresn*, the median of *column* for each distinct value of *expresn*.

In calculating the median, NULL values are ignored.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the median of the values in the column named salary:

```
Median(salary)
```

This expression returns the median of the values in the column named salary of group 1:

```
Median(salary for group 1)
```

This expression returns the median of the values in column 5 on the current page:

```
Median(#5 for page)
```

This computed field returns Above Median if the median salary for the page is greater than the median for the report. The expression is entered on a single line:

```
If(Median(salary for page) > Median(salary), "Above
Median", " ")
```

This expression for a graph value sets the data value to the median value of the sale_price column:

```
Median(sale_price)
```

This expression for a graph value entered in the Graph Data window sets the data value to the median value of the sale_price column for the entire graph:

```
Median(sale_price for graph)
```

Assume the report or DataWindow displays the order number, amount, and line items for each order. This computed field returns the median of the order amount for the distinct order numbers:

```
Median(order_amt for all DISTINCT order_nbr)
```

See also

Avg
Mode

Mid

Description

Obtains a specified number of characters from a specified position in a string.

Syntax

Mid (*string*, *start* {, *length*})

Parameter	Description
string	The string from which you want characters returned.
start	A long specifying the position of the first character you want returned. (The position of the first character of the string is 1.)
length (optional)	A long whose value is the number of characters you want returned. If you do not enter *length* or if *length* is greater than the number of characters to the right of *start*, Mid returns the remaining characters in the string.

Return value

String. Returns characters specified in *length* of *string* starting at character *start*. If *start* is greater than the number of characters in *string*, the Mid function returns the empty string (""). If *length* is greater than the number of characters remaining after the *start* character, Mid returns the remaining characters. The return string is not filled with spaces to make it the specified length.

Examples

This expression returns "":

```
Mid("BABE RUTH", 40, 5)
```

This expression returns BE RUTH:

```
Mid("BABE RUTH", 3)
```

This expression in a computed field returns ACCESS DENIED if the 4th character in the column password is not R:

```
If(Mid(password, 4,1) = "R", "ENTER", "ACCESS
DENIED")
```

To pass this validation rule the fourth character in the column password must be 6:

```
Mid(password, 4,1) = "6"
```

See also

Mid in Chapter 1, "PowerScript Functions"

Min

Description

Determines the minimum value in the specified column.

Syntax

Min (*column* {for *range* { DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the minimum value. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The minimum value of *column*. ♦ Crosstab — (Crosstabs only) The minimum value of *column* in the crosstab. ♦ Graph — (Graphs only) The minimum value of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The minimum value of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The minimum value of *column* on a page.
DISTINCT (optional)	Causes Min to consider only the distinct values in *column* when determining the minimum value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of the column. Returns the minimum value in the rows of *column*. If you specify *range*, Min returns the minimum value in the rows of *column* in *range*.

Usage

If you specify *range*, Min determines the minimum value in *column* in *range*. If you specify DISTINCT, Min returns the minimum distinct value in *column*, or if you specify *expresn*, the minimum distinct value in *column* where the value of *expresn* is distinct.

NULL values are ignored and are not considered in determining the minimum.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the minimum value in the column named age in group 2:

```
Min(age for group 2)
```

This expression returns the minimum of the values in column 3 on the page:

```
Min(#3 for page)
```

Assume the report or DataWindow displays the order number, amount, and line items for each order. This computed field returns the minimum of the order amount for the distinct order numbers:

```
Min(order_amt for all DISTINCT order_nbr)
```

See also

Max
Min in Chapter 1, "PowerScript Functions"

Minute

Description

Obtains the number of minutes in the minutes portion of a time value.

Syntax

Minute (*time*)

Parameter	Description
time	The time value from which you want the minutes

Return value

Integer. Returns the minutes portion of *time* (00 to 59).

Example

This expression returns 1:

```
Minute(19:01:31)
```

See also

Hour
Second
Minute in Chapter 1, "PowerScript Functions"

Mod

Description

Obtains the remainder (modulus) of a division operation.

Syntax

Mod (*x*, *y*)

Parameter	Description
x	The number you want to divide by *y*
y	The number you want to divide into *x*

Return value

The data type of *x* or *y*, whichever data type is more precise.

Examples

This expression returns 2:

```
Mod(20, 6)
```

This expression returns 1.5:

```
Mod(25.5, 4)
```

This expression returns 2.5:

```
Mod(25, 4.5)
```

See also

Mod in Chapter 1, "PowerScript Functions"

Mode

Description

Calculates the mode of the values of the column. The mode is the most frequently occurring value.

Syntax

Mode (*column* { for *range* { DISTINCT { *expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the mode of the data values. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The mode of all rows in *column*. ♦ Crosstab — (Crosstabs only) The mode of all rows in *column* in the crosstab. ♦ Graph — (Graphs only) The mode of all rows in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The mode value of all rows in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The mode of the rows in *column* on a page.
DISTINCT (optional)	Causes Mode to consider only the distinct values in *column* when determining the mode. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The numeric data type of the column. Returns the mode of the values of the rows in *range* if it succeeds and *-1* if an error occurs.

Usage

If you specify *range*, Mode returns the mode of *column* in *range*. If you specify DISTINCT, Mode returns the mode of the distinct values in *column*, or if you specify *expresn*, the mode of *column* for each distinct value of *expresn*.

In calculating the mode, NULL values are ignored.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the mode of the values in the column named salary:

```
Mode(salary)
```

This expression returns the mode of the values for group 1 in the column named salary:

```
Mode(salary for group 1)
```

This expression returns the mode of the values in column 5 on the current page:

```
Mode(#5 for page)
```

This computed field returns Above Mode if the mode of the salary for the page is greater than the mode for the report. The expression is entered on a single line:

```
If(Mode(salary for page) > Mode(salary), "Above
Mode", " ")
```

This expression for a graph value sets the data value to the mode of the sale_price column:

```
Mode(sale_price)
```

This expression for a graph value entered in the Graph Data window sets the data value to the mode of the sale_price column for the entire graph:

```
Mode(sale_price for graph)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the mode of the order amount for the distinct order numbers:

```
Mode(order_amt for all DISTINCT order_nbr)
```

See also

Avg
Median

Month

Description

Determines the month of a date value.

Syntax

Month (*date*)

Parameter	Description
date	The date from which you want the month

Return value

Integer. Returns an integer (1 to 12) whose value is the month portion of *date*.

Examples

This expression returns 1:

```
Month(1994-01-31)
```

This expression for a computed column returns Wrong Month if the month in the column expected_grad_date is not 6. In the painter, you type the expression on a single line:

```
If(Month(expected_grad_date) = 6, "June", "Wrong
Month")
```

This validation rule expression checks that the value of the month in the date in the column expected_grad_date is 6:

```
Month(expected_grad_date) = 6
```

See also

Day
Date
Year
Month in Chapter 1, "PowerScript Functions"

Now

Description

Obtains the current time based on the system time of the client machine.

Syntax

Now ( )

Return value

Time. Returns the current time based on the system time of the client machine.

Usage

Use Now to compare a time to the system time or to display the system time on the screen. The timer interval specified for the DataWindow object determines the frequency at which the value of Now is updated. For example, if the timer interval is 1 second, it is updated every second.

Examples

This expression returns the current system time:

```
Now( )
```

This expression sets the column value to 8:00 when the current system time is before 8:00 and to the current time if it is after 8:00:

```
If (Now( ) < 08:00:00, '08:00:00', String(Now( )))
```

See also

If
Today
Now in Chapter 1, "PowerScript Functions"

Number

Description

Converts a string to a number.

Syntax

Number (*string*)

Parameter	Description
string	The string you want returned as a number

Return value

A numeric data type. Returns the contents of *string* as a number. If *string* is not a valid number, Number returns *0*.

Examples

This expression converts the string 24 to a number:

```
Number("24")
```

This expression for a computed field tests whether the value in the age column is greater than 55 and if so displays N/A; otherwise, it displays the value in age:

```
If(Number(age) > 55, "N/A", age)
```

This validation rule checks that that number the user entered is between 25,000 and 50,000:

```
Number(GetText())>25000 AND Number(GetText())<50000
```

Page

Description

Determines the number of the current page.

Syntax

Page ( )

Return value

Integer. Returns the number of the current page.

> **Calculating the page count**
> The vertical size of the paper less the top and bottom margins is used to calculate the page count. When the print orientation is landscape, the vertical size of the paper is the shorter dimension of the paper.

Examples

This expression returns the number of the current page:

```
Page( )
```

This expression for a computed field in the DataWindow's footer band displays a string showing the current page number and the total number of pages in the report. The result has the format **Page** *n* **of** *total*:

```
'Page ' + Page() + ' of ' + PageCount( )
```

PageAcross

Description

Determines the number of the current horizontal page. For example, if a report is twice the width of the preview window and the window is scrolled horizontally to display the portion of the report that was outside the preview, PageAcross will return 2 because the current page is the second horizontal page.

Syntax

PageAcross ( )

Return value

Integer. Returns the number of the current horizontal page if it succeeds and *-1* if an error occurs.

Example

This expression returns the number of the current horizontal page:

```
PageAcross( )
```

PageCount

Description

Determines the total number of pages.

Syntax

PageCount ( )

Return value

Integer. Returns the total number of pages.

> **Calculating the page count**
> The vertical size of the paper less the top and bottom margins is used to calculate the page count. When the print orientation is landscape, the vertical size of the paper is the shorter dimension of the paper.

Example

This expression returns the number of pages:

```
PageCount( )
```

This expression for a computed field in the DataWindow's footer band displays a string showing the current page number and the total number of pages in the report. The result has the format **Page** *n* **of** *total*:

```
'Page ' + Page() + ' of ' + PageCount( )
```

PageCountAcross

Description

Determines the total number of horizontal pages when a report or DataWindow is wider than the preview window.

Syntax

PageCountAcross ( )

Return value

Integer. Returns the total number of horizontal pages if it succeeds and *-1* if an error occurs.

Usage

PageCountAcross applies to Print Preview, not the design preview available in the DataWindow painter.

Examples

This expression returns the number of horizontal pages in the preview window:

```
PageCountAcross( )
```

Percent

Description

Determines the percentage that the current value is of the total of the values in the column.

Syntax

Percent (*column* { for *range* {DISTINCT {*expres1*{, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the value of each row as a percentage of the total of the values of the column. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The percentage for each row of *column*. ♦ Graph — (Graphs only) The percentage for each row of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The percentage for each row of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The percentage for each row of *column* on a page.
DISTINCT (optional)	Causes Percent to consider only the distinct values in *column* when determining the percentage. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

A numeric data type (decimal, double, integer, long, or real). Returns the percentage the current row of *column* is of the total value of the column.

Usage

Usually you use Percent in a column to display the percentage for each row. You can also use Percent in a header or trailer for a group. In the header, it displays the percentage for the first value in the group and in the trailer for the last value in the group.

If you specify *range*, Percent returns the percentage the current row of *column* relative to the total value of *range*. For example, if column 5 is salary, Percent(#5 for group 1) is equivalent to: salary/(Sum(Salary for group 1).

If you specify DISTINCT, Percent returns the percent the distinct value in *column* is of the total value of *column*, or if you specify expres***n***, the percentage the value in *column* in the row in which the value of *expres**n*** is distinct, is of the total for *column*.

Formatting the percent value
The percentage is displayed as a decimal value unless you specify a format for the result. A display format can be part of the computed field's definition.

NULL values are ignored and are not considered in the calculation.

Not in validation rules, filter expressions, or crosstabs
You cannot use Percent or other aggregate functions in validation rules or filter expressions. Percent does not work for crosstabs; specifying "for crosstab" as a range is not available for Percent.

Examples

This expression returns the value of each row in the column named salary as a percentage of the total of salary:

```
Percent(salary)
```

This expression returns the value of each row in the column named cost as a percentage of the total of cost in group 2:

```
Percent(cost for group 2)
```

This expression entered in the Values box in the Graph Data window returns the value of each row in the qty_ordered as a percentage of the total for the column in the graph:

```
Percent(qty_ordered for graph)
```

Assume the report or DataWindow displays the order number, amount, and line items for each order. This computed field returns the order amount as a percentage of the total order amount for the distinct order numbers:

```
Percent(order_amt for all DISTINCT order_nbr)
```

See also

CumulativePercent

Pi

Description

Multiplies pi by a specified number.

Syntax

Pi (*n*)

Parameter	Description
n	The number you want to multiply by pi (3.14159265358979323...)

Return value

Double. Returns the result of multiplying *n* by pi if it succeeds and *-1* if an error occurs.

Usage

Use Pi to convert angles to and from radians.

Return value

Double. Returns the result of multiplying *n* by pi if it succeeds and -1 if an error occurs.

Examples

This expression returns pi:

```
Pi(1)
```

Both these expressions return the area of a circle with the radius Rad:

```
Pi(1) * Rad^2
```

```
Pi(Rad^2)
```

This expression computes the cosine of a 45-degree angle:

```
Cos(45.0 * (Pi(2)/360))
```

See also

Cos
Sin
Tan
Pi in Chapter 1, "PowerScript Functions"

Pos

Description

Finds one string within another string.

Syntax

Pos (*string1*, *string2* {, start})

Parameter	Description
string1	The string in which you want to find *string2*.
string2	The string you want to find in *string1*.
start (optional)	A long indicating where the search will begin in *string1*. The default is 1.

Return value

Long. Returns a long whose value is the starting position of the first occurrence of *string2* in *string1* after the position specified in *start*. If *string2* is not found in *string1* or if *start* is not within *string1*, Pos returns *0*.

Usage

The Pos function is case sensitive.

Examples

This expression returns 6:

```
Pos("BABE RUTH", "RU")
```

This expression returns 1:

```
Pos("BABE RUTH", "B")
```

This expression returns 0 because the case does not match:

```
Pos("BABE RUTH", "be")
```

This expression returns 0 because it starts searching at position 5, after the occurrence of BE:

```
Pos("BABE RUTH", "BE", 5)
```

See also

Left
Mid
Right
Pos in Chapter 1, "PowerScript Functions"

ProfileInt

Description

Obtains the integer value of a setting in the specified profile file.

Syntax

ProfileInt (*filename, section, key, default*)

Parameter	Description
filename	A string whose value is the name of the profile file. If you do not specify a full path, ProfileInt uses the operating system's standard file search order to find the file.
section	A string whose value is the name of a group of related values in the profile file. In the file, section names are in square brackets. Do not include the brackets in *section*. *Section* is not case-sensitive.
key	A string specifying the setting name in *section* whose value you want. The setting name is followed by an equal sign in the file. Do not include the equal sign in *key*. *Key* is not case-sensitive.
default	An integer value that ProfileInt will return if *filename* is not found, if *section* or *key* does not exist in *filename*, or if the value of *key* cannot be converted to an integer.

Return value

Integer. Returns *default* if *filename* is not found, *section* is not found in *filename*, or *key* is not found in *section*, or the value of *key* is not an integer. Returns *-1* if an error occurs.

Usage

Use ProfileInt or ProfileString to get configuration settings from a profile file that you've designed for your application.

You can use SetProfileString to change values in the profile file to customize your application's configuration during execution. Before you make changes, you can use ProfileInt and ProfileString to obtain the original settings so you can restore them when the user exits the application, if desired.

Example

This example uses the following PROFILE.INI file:

```
[MyApp]
Maximized=1
[security]
Class = 7
```

This expression tries to return the integer value of the keyword Minimized in section MyApp of file C:\PROFILE.INI. It returns 3 if there is no MyApp section or no Minimized keyword in the MyApp section. Based on the sample file above, it returns 3:

```
ProfileInt("C:\PROFILE.INI", "MyApp", "minimized", 3)
```

See also

ProfileString
ProfileInt in Chapter 1, "PowerScript Functions"

ProfileString

Description

Obtains the string value of a setting in the specified profile file.

Syntax

ProfileString (*filename, section, key, default*)

Parameter	Description
filename	A string whose value is the name of the profile file. If you do not specify a full path, ProfileString uses the operating system's standard file search order to find the file.
section	A string whose value is the name of a group of related values in the profile file. In the file, section names are in square brackets. Do not include the brackets in *section*. *Section* is not case-sensitive.
key	A string specifying the setting name in *section* whose value you want. The setting name is followed by an equal sign in the file. Do not include the equal sign in *key*. *Key* is not case-sensitive.
default	A string value that ProfileString will return if *filename* is not found, if *section* or *key* does not exist in *filename*, or if the value of *key* cannot be converted to an integer.

Return value

String, with a maximum length of 4096 characters. Returns the string from *key* within *section* within *filename*. If *filename* is not found, *section* is not found in *filename*, or *key* is not found in *section*, ProfileString returns *default*. If an error occurs, it returns the empty string ("").

Usage

Use ProfileInt or ProfileString to get configuration settings from a profile file that you've designed for your application.

You can use SetProfileString to change values in the profile file to customize your application's configuration during execution. Before you make changes, you can use ProfileInt and ProfileString to obtain the original settings so you can restore them when the user exits the application, if desired.

Example

This example uses the following section in PROFILE.INI file:

```
[Employee]
Name="Smith"
[Dept]
Name="Marketing"
```

This expression, typed on a single line in the painter, returns the string for the keyword Name in section Employee in file C:\PROFILE.INI. It returns None if the section or keyword does not exist. In this case it returns Smith:

```
ProfileString("C:\PROFILE.INI", "Employee", "Name",
"None")
```

See also

ProfileInt
ProfileString in Chapter 1, "PowerScript Functions"

Rand

Description

Obtains a random whole number between 1 and a specified upper limit.

Syntax

Rand (*n*)

Parameter	Description
n	The upper limit of the range of random numbers you want returned. The lower limit is always 1. The upper limit cannot exceed 32,767.

Return value

A numeric data type, the data type of *n*. Returns a random whole number between 1 and *n*.

Usage

The sequence of numbers generated by repeated calls to the Rand function is a computer-generated pseudo-random sequence. You can control whether the sequence is different each time your application runs by calling the Randomize function to initialize the random number generator.

Example

This expression returns a random whole number between 1 and 10:

```
Rand(10)
```

See also

Rand in Chapter 1, "PowerScript Functions"

Real

Description

Converts a string value to a real data type.

Syntax

Real (*string*)

Parameter	Description
string	The string whose value you want to convert to a real

Return value

Real. Returns the contents of a string as a real. If string is not a valid number, Real returns *0*.

Examples

This expression converts 24 to a real:

```
Real("24")
```

This expression returns the value in the column temp_text as a real:

```
Real(temp_text)
```

See also

Real in Chapter 1, "PowerScript Functions"

RelativeDate

Description

Obtains the date that occurs a specified number of days after or before another date.

Syntax

RelativeDate (*date, n*)

Parameter	Description
date	A date value
n	An integer indicating the number of days

Return value

Date. Returns the date that occurs *n* days after *date* if *n* is greater than 0. Returns the date that occurs *n* days before *date* if *n* is less than 0.

Examples

This expression returns 1990-02-10:

```
RelativeDate(1990-01-31, 10)
```

This expression returns 1990-01-21:

```
RelativeDate(1990-01-31, -10)
```

See also

DaysAfter
RelativeDate in Chapter 1, "PowerScript Functions"

RelativeTime

Description

Obtains a time that occurs a specified number of seconds after or before another time within a 24-hour period.

Syntax

RelativeTime (*time, n*)

Parameter	Description
time	A time value
n	A long number of seconds

Return value

Time. Returns the time that occurs *n* seconds after *time* if *n* is greater than 0. Returns the time that occurs *n* seconds before *time* if *n* is less than 0. The maximum return value is 23:59:59.

Examples

This expression returns 19:01:41:

```
RelativeTime(19:01:31, 10)
```

This expression returns 19:01:21:

```
RelativeTime(19:01:31, -10)
```

See also

SecondsAfter
RelativeTime in Chapter 1, "PowerScript Functions"

Replace

Description

Replaces a portion of one string with another.

Syntax

Replace (*string1, start, n, string2*)

Parameter	Description
string1	The string in which you want to replace characters with *string2.*
start	A long whose value is the number of the first character you want replaced. (The first character in the string is number 1.)
n	A long whose value is the number of characters you want to replace.
string2	The string that will replace characters in *string1*. The number of characters in *string2* can be greater than, equal to, or less than the number of characters you are replacing.

Return value

String. Returns the string with the characters replaced if it succeeds and the empty string ("") if it fails.

Usage

If the start position is beyond the end of the string, Replace appends *string2* to *string1*. If there are fewer characters after the start position than specified in *n,* Replace replaces all the characters to the right of character *start*.

If *n* is zero, then, in effect, Replace inserts *string2* into *string1*.

Examples

This expression changes the last two characters of the string David to e to make it Dave:

```
Replace("David", 4, 2, "e")
```

This expression returns BABY RUTH:

```
Replace("BABE RUTH", 1, 4, "BABY")
```

This expression returns Closed for the Winter:

```
Replace("Closed for Vacation", 12, 8, "the Winter")
```

See also

Replace in Chapter 1, "PowerScript Functions"

Right

Description

Obtains a specified number of characters from the end of a string.

Syntax

Right (*string, n*)

Parameter	Description
string	The string from which you want characters returned
n	A long whose value is the number of characters you want returned from the right end of *string*

Return value

String. Returns the rightmost *n* characters in *string* if it succeeds and the empty string ("") if an error occurs.

If *n* is greater than or equal to the length of the string, Right returns the entire string. It does not add spaces to make the return value's length equal to *n*.

Examples

This expression returns RUTH:

```
Right("BABE RUTH", 4)
```

This expression returns BABE RUTH:

```
Right("BABE RUTH", 75)
```

See also

Left
Mid
Pos
Right in Chapter 1, "PowerScript Functions"

RightTrim

Description

Removes spaces from the end of a string.

Syntax

RightTrim (*string*)

Parameter	Description
string	The string you want returned with trailing blanks deleted

Return value

String. Returns a copy of *string* with trailing blanks deleted if it succeeds and the empty string ("") if an error occurs.

Example

This expression returns RUTH:

```
RightTrim("RUTH    ")
```

See also

LeftTrim
Trim
RightTrim in Chapter 1, "PowerScript Functions"

Round

Description

Rounds a number to the specified number of decimal places.

Syntax

Round (*x, n*)

Parameter	Description
x	The number you want to round
n	The number of decimal places to which you want to round *x*

Return value

Decimal. If *n* is positive, returns *x* rounded to the specified number of decimal places. If *n* is negative, returns *x* rounded to (*-n*+1) places before the decimal point. Returns *-1* if it fails.

Examples

This expression returns 9.62:

```
Round(9.624, 2)
```

This expression returns 9.63:

```
Round(9.625, 2)
```

This expression returns 9.600:

```
Round(9.6, 3)
```

This expression returns -9.63:

```
Round(-9.625, 2)
```

This expression returns -10:

```
Round(-9.625, -1)
```

See also

Ceiling
Int
Truncate
Round in Chapter 1, "PowerScript Functions"

RowCount

Description

Obtains the number of rows that are currently available in the DataWindow's primary buffer.

Syntax

RowCount ( )

Return value

Long. Returns the number of rows that are currently available, *0* if no rows are currently available, and *-1* if an error occurs.

Example

This expression in a computed field returns a warning if no data exists and the number of rows if there is data:

```
If(RowCount( ) = 0, "No Data", String(RowCount()))
```

See also

RowCount in Chapter 1, "PowerScript Functions"

RowHeight

Description

Reports the height of a row associated with a band in a DataWindow control.

Syntax

RowHeight ( )

Return value

Long. Returns the height of the row in the units specified for the DataWindow object if it succeeds, and *-1* if an error occurs.

Usage

When you call RowHeight in bands other than the detail band, it reports on a row in the detail band. See GetRow for a table specifying which row is associated with each band for reporting purposes.

Example

This expression for a computed field in the detail band displays the height of each row:

```
RowHeight( )
```

See also

GetRow

Second

Description

Obtains the number of seconds in the seconds portion of a time value.

Syntax

Second (*time*)

Parameter	Description
time	The time value from which you want the seconds

Return value

Integer. Returns the seconds portion of *time* (00 to 59).

Example

This expression returns 31:

```
Second(19:01:31)
```

See also

Hour
Minute
Second in Chapter 1, "PowerScript Functions"

SecondsAfter

Description

Determines the number of seconds one time occurs after another.

Syntax

SecondsAfter (*time1*, *time2*)

Parameter	Description
time1	A time value that is the start time of the interval being measured
time2	A time value that is the end time of the interval

Return value

Long. Returns the number of seconds *time2* occurs after *time1*. If *time2* occurs before *time1*, SecondsAfter returns a negative number.

Examples

This expression returns 15:

```
SecondsAfter(21:15:30, 21:15:45)
```

This expression returns -15:

```
SecondsAfter(21:15:45, 21:15:30)
```

This expression returns 0:

```
SecondsAfter(21:15:45, 21:15:45)
```

See also

DaysAfter
SecondsAfter in Chapter 1, "PowerScript Functions"

Sign

Description

Determines the sign of a number. Sign reports whether a number is negative, zero, or positive.

Syntax

Sign (*n*)

Parameter	Description
n	The number for which you want to determine the sign

Return value

Integer. Returns a number (-1, 0, or 1) indicating the sign of *n*.

Examples

This expression returns 1 (the number is positive):

```
Sign(5)
```

This expression returns 0 (zero has no sign):

```
Sign(0)
```

This expression returns -1 (the number is negative):

```
Sign(-5)
```

See also

Sign in Chapter 1, "PowerScript Functions"

Sin

Description

Calculates the sine of an angle.

Syntax

Sin (*n*)

Parameter	Description
n	The angle (in radians) for which you want the sine

Return value

Double. Returns *1* if it succeeds and *-1* if an error occurs.

Examples

This expression returns .8414709848078965:

```
Sin(1)
```

This expression returns 0:

```
Sin(0)
```

This expression returns 0:

```
Sin(pi(1))
```

See also

Cos
Pi
Tan
Sin in Chapter 1, "PowerScript Functions"

Small

Description

Finds a small value at a specified ranking in a column (for example, third smallest, fifth smallest) and returns the value of another column or expression based on the result.

Syntax

Small (*returnexp*, *column*, *nbottom* {FOR *range* { DISTINCT { *expres1* {, *expres2* {, ...}}}}})

Parameter	Description
returnexp	The value you want returned when the small value is found. *Returnexp* includes a reference to a column, but not necessarily the column that is being evaluated for the small value, so that a value is returned from the same row that contains the small value.
column	The column that contains the small value you are searching for. *Column* can be a column name or a column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
nbottom	The relationship of the small value to the column's smallest value. For example, when *nbottom* is 2, Small finds the second smallest value.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The small value in *column*. ♦ Crosstab — (Crosstabs only) The small value in *column* in the crosstab. ♦ Graph — (Graphs only) The small value in *column* for the graph. This value for *range* has effect only when you specify Page in the Rows in the Graph Data window. ♦ GroupNbr — The small value in *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The small value in *column* on a page.
DISTINCT (optional)	Causes Small to consider only the distinct values in *column* when determining the small value. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The data type of *returnexp*. Returns the *nbottom* smallest value if it succeeds and *-1* if an error occurs.

Usage

If you specify *range*, Small returns the value in *returnexp* when the value in *column* is the *nbottom* smallest value in *range*. If you specify DISTINCT, Small returns *returnexp* when the value in *column* is the *nbottom* smallest value of the distinct values in *column*, or if you specify *expresn*, the *nbottom* smallest for each distinct value of *expresn*.

Tip
If you don't need a return value from another column and you want to find the smallest value (*nbottom* = 1), use Min; it is faster.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Example

These expressions return the names of the salespeople with the three smallest sales (sum_sales is the sum of the sales for each salesperson) in group 2, which might be the salesregion group. Note that sum_sales contains the values being compared, but Small returns a value in the name column:

```
Small(name, sum_sales, 1 for group 2)
Small(name, sum_sales, 2 for group 2)
Small(name, sum_sales, 3 for group 2)
```

This example reports the salesperson with the third smallest sales, considering only the first entry for each person:

```
Small(name, sum_sales, 3 for all DISTINCT sum_sales)
```

See also

Large

Space

Description

Builds a string of the specified length whose value consists of spaces.

Syntax

Space (*n*)

Parameter	Description
n	A long whose value is the length of the string you want filled with spaces

Return value

String. Returns a string filled with *n* spaces if it succeeds and the empty string ("") if an error occurs.

Example

This expression for a computed field returns 10 spaces in the computed field if the value of the rating column is Top Secret; otherwise, it returns the value in rating:

```
If(rating = "Top Secret", Space(10), rating)
```

See also

Fill
Space in Chapter 1, "PowerScript Functions"

Sqrt

Description

Calculates the square root of a number.

Syntax

Sqrt (*n*)

Parameter	Description
n	The number for which you want the square root

Return value

Double. Returns the square root of *n*.

Usage

Sqrt(*n*) is the same as *n*^.5.

Taking the square root of a negative number causes an execution error.

Examples

This expression returns 1.414213562373095:

```
Sqrt(2)
```

This expression results in an error at execution time:

```
Sqrt(-2)
```

See also

Sqrt in Chapter 1, "PowerScript Functions"

StDev

Description

Calculates an estimate of the standard deviation for the specified column. Standard deviation is a measurement of how widely values vary from average.

Syntax

StDev (*column* {for *range* { DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which for which you want an estimate for the standard deviation of the values in the rows. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The estimate for the standard deviation of the values in the rows of *column*. ♦ Crosstab — (Crosstabs only) The estimate for the standard deviation of the values in the rows of *column* in the crosstab. ♦ Graph — (Graphs only) The estimate for the standard deviation of the values in the rows of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window.

Parameter	Description
	♦ GroupNbr — The estimate for the standard deviation of the values in the rows of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The estimate for the standard deviation of the values in the rows of *column* on a page.
DISTINCT (optional)	Causes StDev to consider only the distinct values in *column* when determining the standard deviation. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

Double. Returns an estimate of the standard deviation for *column*.

Usage

If you specify *range*, StDev returns an estimate for the standard deviation of *column* within *range*. If you specify DISTINCT, StDev returns an estimate of the standard deviation for the distinct values in *column* or if you specify *expresn*, the estimate of the standard deviation of the rows in *column* where the value of *expresn* is distinct.

Estimating or calculating actual standard deviation
StDev assumes that the values in *column* are a sample of the values in the rows in the column in the database table. If you selected all the rows in the column in the DataWindow's SELECT statement, use StDevP to compute the standard deviation of the population.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

These examples all assume that the SELECT statement did not retrieve all of the rows in the database table. StDev is intended to work with a subset of rows, which is a sample of the full set of data.

This expression returns an estimate for standard deviation of the values in the column named salary:

```
StDev(salary)
```

This expression returns an estimate for standard deviation of the values in the column named salary in group 1:

```
StDev(salary for group 1)
```

This expression returns an estimate for standard deviation of the values in column 4 on the page:

```
StDev(#4 for page)
```

This expression entered in the Values box in the Graph Data window returns an estimate for standard deviation of the values in qty_used column in the graph:

```
StDev(qty_used for graph)
```

This expression for a computed field in a crosstab returns the estimate for standard deviation of the values in qty_ordered column in the crosstab:

```
StDev(qty_ordered for crosstab)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the estimated standard deviation of the order amount for the distinct order numbers:

```
StDev(order_amt for all DISTINCT order_nbr)
```

StDevP

Description

Calculates the standard deviation for the specified column. Standard deviation is a measurement of how widely values vary from average.

Syntax

StDevP (*column* {for *range* { DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the standard deviation of the values in the rows. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.

Parameter	Description
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The standard deviation of the values in the rows of *column*. ♦ Crosstab — (Crosstabs only) The standard deviation of the values in the rows of *column* in the crosstab. ♦ Graph — (Graphs only) The standard deviation of the values in the rows of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The standard deviation of the values in the rows of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The standard deviation of the values in the rows of *column* on a page.
DISTINCT (optional)	Causes StDevP to consider only the distinct values in *column* when determining the standard deviation. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

Double. Returns the standard deviation for *column*.

Usage

If you specify *range*, StDevP returns the standard deviation for *column* within *range*. If you specify DISTINCT, StDev returns an estimate of the standard deviation for the distinct values in *column* or if you specify *expresn*, the estimate of the standard deviation of the rows in *column* where the value of *expresn* is distinct.

Estimating or calculating actual standard deviation
StDevP assumes that the values in *column* are the values in all the rows in the column in the database table. If you did not select all rows in the column in the DataWindow's SELECT statement, use StDev to compute an estimate of the standard deviation of a sample.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

These examples all assume that the SELECT statement retrieved all rows in the database table. StDevP is intended to work with a full set of data, not a subset.

This expression returns the standard deviation of the values in the column named salary:

```
StDevP(salary)
```

This expression returns the standard deviation of the values in group 1 in the column named salary:

```
StDevP(salary for group 1)
```

This expression returns the standard deviation of the values in column 4 on the page:

```
StDevP(#4 for page)
```

This expression entered in the Values box in the Graph Data window returns the standard deviation of the values in qty_ordered column in the graph:

```
StDevP(qty_ordered for graph)
```

This expression for a computed field in a crosstab returns the standard deviation of the values in qty_ordered column in the crosstab:

```
StDevP(qty_ordered for crosstab)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the standard deviation of the order amount for the distinct order numbers:

```
StDevP(order_amt for all DISTINCT order_nbr)
```

String

Description

Formats data as a string according to a specified display format mask. You can convert and format date, DateTime, numeric, and time data. You can also apply a display format to a string.

Syntax

String (*data*, { *format* })

Parameter	Description
data	The data you want returned as a string with the specified formatting. *Data* can have a date, DateTime, numeric, time, or string data type.
format (optional)	A string of the display masks you want to use to format the data. The masks consist of formatting information specific to the data type of *data*. If *data* is type string, *format* is required. The format string can consist of more than one mask, depending on the data type of *data*. Each mask is separated by a semicolon. See Usage for details on each data type.

Return value

String. Returns *data* in the specified format if it succeeds and the empty string ("") if the data type of *data* does not match the type of display mask specified or *format* is not a valid mask.

Usage

For date, DateTime, numeric, and time data, PowerBuilder uses the system's default format for the returned string if you don't specify a format. For numeric data, the default format is the [General] format.

For string data, a display format mask is required. (Otherwise, the function would have nothing to do.)

The format can consist of one or more masks:

- Formats for date, DateTime, string, and time data can include one or two masks. The first mask is the format for the data; the second mask is the format for a null value.
- Formats for numeric data can have up to four masks. A format with a single mask handles both positive and negative data. If there are additional masks, the first mask is for positive values, and the additional masks are for negative, zero, and NULL values.

A format can include color specifications, which are displayed in the DataWindow. Note that when you use the String function in a script, rather than in the DataWindow painter, color specifications have no effect.

If the display format doesn't match the data type, PowerBuilder will try to apply the mask, which can produce unpredictable results.

For more information on specifying display formats, see the *User's Guide*.

Examples

This expression returns Jan 31, 1998:

```
String(1998-01-31, "mmm dd, yyyy")
```

This expression returns Jan 31, 1998 6 hrs and 8 min. The expression is entered on a single line:

```
String(1998-01-31 06:08:00, 'mmm dd, yyyy h "hrs
and" m "min"')
```

This expression:

```
String(nbr, "0000;(000);****;empty")
```

returns:

0123	if nbr is 123
(123)	if nbr is -123
****	if nbr is 0
empty	if nbr is NULL

This expression returns A-B-C:

```
String("ABC", "@-@-@")
```

This expression returns A*B:

```
String("ABC", "@*@")
```

This expression returns ABC:

```
String("ABC", "@@@")
```

This expression returns a space:

```
String("ABC", " ")
```

This expression returns 6 hrs and 8 min:

```
String(06:08:02,'h "hrs and" m "min"')
```

This expression returns 08:06:04 pm:

```
String(20:06:04, "hh:mm:ss am/pm")
```

This expression returns 8:06:04 am:

```
String(08:06:04, "h:mm:ss am/pm")
```

This expression returns 6:11:25.300000:

```
String(6:11:25.300000, "h:mm:ss.ffffff")
```

See also

String in Chapter 1, "PowerScript Functions"

Sum

Description

Calculates the sum of the values in the specified column.

Syntax

Sum (*column* {for *range* {DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which you want the sum of the data values. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) The sum of the values of the rows of *column*. ♦ Crosstab — (Crosstabs only) The sum of the values in the rows of *column* in the crosstab. ♦ Graph — (Graphs only) The sum of the values in the rows of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The sum of the values of the rows of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The sum of the values of *column* on a page.
DISTINCT (optional)	Causes Sum to consider only the distinct values in *column* when determining the sum. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

The appropriate numeric data type. Returns the sum of the data values in *column*.

Usage

If you specify *range*, Sum returns the sum of the values in *column* within *range*. If you specify DISTINCT, Sum returns the sum of the distinct values in *column*, or if you specify *expresn*, the sum of the values of *column* where the value of *expresn* is distinct.

NULL values are ignored and are not included in the calculation.

> **Not in validation rules or filter expressions**
> You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

This expression returns the sum of the values in group 1 in the column named salary:

```
Sum(salary for group 1)
```

This expression returns the sum of the values in column 4 on the page:

```
Sum(#4 for page)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the sum of the order amount for the distinct order numbers:

```
Sum(order_amt for all DISTINCT order_nbr)
```

Tan

Description

Calculates the tangent of an angle.

Syntax

Tan (*n*)

Parameter	Description
n	The angle (in radians) for which you want the tangent

Return value

Double. Returns the tangent of *n* if it succeeds and *-1* if an error occurs.

Examples

Both these expressions return 0:

```
Tan(0)
Tan(Pi(1))
```

This expression returns 1.55741:

```
Tan(1)
```

See also

Cos
Pi
Sin
Tan in Chapter 1, "PowerScript Functions"

Time

Description

Converts a string to a time data type.

Syntax

Time (*string*)

Parameter	Description
string	A string containing a valid time (such as 8AM or 10:25) that you want returned as a time data type. Only the hour is required; you do not have to include the minutes, seconds, or microseconds of the time or AM or PM. The default value for minutes and seconds is 00 and for microseconds is 000000. AM or PM is determined automatically.

Return value

Time. Returns the time in *string* as a time data type. If *string* does not contain a valid time, Time returns 00:00:00.

Examples

This expression returns the time data type for 45 seconds before midnight (23:59:15):

```
Time("23:59:15")
```

This expression for a computed field returns the value in the time_received column as a value of type time if time_received is not the empty string. Otherwise, it returns 00:00:00:

```
If(time_received = "" ,00:00:00,
Time(time_received))
```

This example is similar to the previous one, except that it return 00:00:00 if time_received contains a NULL value. The expression is entered as a single line:

```
If(IsNull(time_received), 00:00:00,
Time(time_received))
```

See also

Time in Chapter 1, "PowerScript Functions"

Today

Description

Obtains the system date.

Syntax

Today ( )

Return value

DateTime. Returns the current system DateTime.

Example

This expression returns the current system date:

```
Today( )
```

See also

Now
Today in Chapter 1, "PowerScript Functions"

Trim

Description

Removes leading and trailing spaces from a string.

Syntax

Trim (*string*)

Parameter	Description
string	The string you want returned with leading and trailing spaces deleted

Return value

String. Returns a copy of *string* with all leading and trailing spaces deleted if it succeeds and the empty string ("") if an error occurs.

Usage

Trim is useful for removing spaces that a user may have typed before or after newly entered data.

Examples

This expression returns BABE RUTH:

```
Trim("      BABE RUTH       ")
```

See also

LeftTrim
RightTrim
Trim in Chapter 1, "PowerScript Functions"

Truncate

Description

Truncates a number to the specified number of decimal places.

Syntax

Truncate (*x, n*)

Parameter	Description
x	The number you want to truncate
n	The number of decimal places to which you want to truncate *x*

Return value

The data type of *n*. If *n* is positive, returns *x* truncated to the specified number of decimal places. If *n* is negative, returns *x* truncated to (*-n*+1) places before the decimal point. Returns *-1* if it fails.

Examples

This expression returns 9.2:

```
Truncate(9.22, 1)
```

This expression returns 9.2:

```
Truncate(9.28, 1)
```

This expression returns 9:

```
Truncate(9.9, 0)
```

This expression returns –9.2:

```
Truncate(–9.29, 1)
```

This expression returns 0:

```
Truncate(9.2, –1)
```

This expression returns 50:

```
Truncate(54, –1)
```

See also

Ceiling
Int
Round
Truncate in Chapter 1, "PowerScript Functions"

Upper

Description

Converts all the characters in a string to uppercase.

Syntax

Upper (*string*)

Parameter	Description
string	The string you want to convert to uppercase letters

Return value

String. Returns *string* with lowercase letters changed to uppercase if it succeeds and the empty string ("") if an error occurs.

Example

This expression returns BABE RUTH:

```
Upper("Babe Ruth")
```

See also

Lower
Upper in Chapter 1, "PowerScript Functions"

Var

Description

Calculates an estimate of the variance for the specified column.

Syntax

Var (*column* {for *range* {DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which for which you want an estimate for the variance of the values in the rows. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values for *range* are: ♦ All — (Default) An estimate for the variance of the values of the rows of *column*. ♦ Crosstab — (Crosstabs only) The estimate for the variance of the values in the rows of *column* in the crosstab. ♦ Graph — (Graphs only) The estimate for the variance of the values in the rows of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window.

Parameter	Description
	♦ GroupNbr — An estimate for the variance of the values of the rows of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — An estimate for the variance of the values of the rows of *column* on a page.
DISTINCT (optional)	Causes Var to consider only the distinct values in *column* when determining the variance. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

Double. Returns an estimate for the variance for *column*. If you specify *group*, Var returns an estimate for the variance for *column* within *group*.

Usage

If you specify *range*, Var returns an estimate for the variance for *column* within *range*. If you specify DISTINCT, VarP returns the variance for the distinct values in *column* or if you specify *expresn*, the estimate for the variance of the rows in *column* where the value of *expresn* is distinct.

Estimating variance or calculating actual variance
Var assumes that the values in *column* are a sample of the values in rows in the column in the database table. If you select all rows in the column in the SELECT statement, use VarP to compute the variance of a population.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

These examples all assume that the SELECT statement did not retrieve all of the rows in the database table. Var is intended to work with a subset of rows, which is a sample of the full set of data.

This expression returns an estimate for the variance of the values in the column named salary:

```
Var(salary)
```

This expression returns an estimate for the variance of the values in the column named salary in group 1:

```
Var(salary for group 1)
```

This expression entered in the Values box in the Graph Data window returns an estimate for the variance of the values in quantity column in the graph:

```
Var(quantity for graph)
```

This expression for a computed field in a crosstab returns an estimate for the variance of the values in the quantity column in the crosstab:

```
Var(quantity for crosstab)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the estimate for the variance of the order amount for the distinct order numbers:

```
Var(order_amt for all DISTINCT order_nbr)
```

VarP

Description

Calculates the variance for the specified column.

Syntax

VarP (*column* {for *range* {DISTINCT {*expres1* {, *expres2* {, ...}}}}})

Parameter	Description
column	The column for which for which you want the variance of the values in the rows. *Column* can be the column name or the column number preceded by a pound sign (#). *Column* can also be an expression that includes a reference to the column. The data type of *column* must be numeric.
for *range* (optional)	If you specify *range*, you must precede it with the keyword *for*. Values *range* are: ♦ All — (Default) The variance of the values of the rows of *column*. ♦ Crosstab — (Crosstabs only) The variance of the values in the rows of *column* in the crosstab.

Parameter	Description
	♦ Graph — (Graphs only) The variance of the values in the rows of *column* for the graph. This value for *range* has effect only when you specify Page in the Rows option in the Graph Data window. ♦ GroupNbr — The variance of the values of the rows of *column* in the specified group. Specify the keyword group followed by the group number. For example: for group 1. ♦ Page — The variance of the values of the rows of *column* displayed on a page.
DISTINCT (optional)	Causes VarP to consider only the distinct values in *column* when determining the variance. For a value of *column*, the first row found with the value is used and other rows that have the same value are ignored.
expresn (optional)	One or more expressions that you want to evaluate to determine distinct rows. *Expresn* can be the name of a column, a function, or an expression.

Return value

Double. Returns the variance for *column*. If you specify *group*, Var returns the variance for *column* within *range*.

Usage

If you specify *range*, VarP returns the variance for *column* within *range*. If you specify DISTINCT, VarP returns the variance for the distinct values in *column* or if you specify *expresn*, the variance of the rows in *column* where the value of *expresn* is distinct.

Estimating variance or calculating actual variance
VarP assumes that the values in *column* are the values in all rows in the column in the database table. If you did not select all the rows in the column in the SELECT statement, use Var to compute an estimate of the variance of a sample.

Not in validation rules or filter expressions
You cannot use this or other aggregate functions in validation rules or filter expressions.

Examples

These examples all assume that the SELECT statement retrieved all rows in the database table. VarP is intended to work with a full set of data, not a subset.

This expression returns the variance of the values in the column named salary:

```
VarP(salary)
```

This expression returns the variance of the values in group 1 in the column named salary:

```
VarP(salary for group 1)
```

This expression returns the variance of the values in column 4 on the page:

```
VarP(#4 for page)
```

This expression entered in the Values box in the Graph Data window returns the variance of the values in quantity column in the graph:

```
VarP(quantity for graph)
```

This expression for a computed field in a crosstab returns the variance of the values in quantity column in the crosstab:

```
VarP(quantity for crosstab)
```

Assume the report displays the order number, amount, and line items for each order. This computed field returns the variance of the order amount for the distinct order numbers:

```
VarP(order_amt for all DISTINCT order_nbr)
```

WordCap

Description

Sets the first letter of each word in a string to a capital letter and all other letters to lowercase (for example, ROBERT E. LEE would be Robert E. Lee).

Syntax

WordCap (*string*)

Parameter	Description
string	A string or expression that evaluates to a string that you want to display with initial capital letters (for example, Monday Morning)

Return value

String. Returns *string* with the first letter of each word set to uppercase and the remaining letters lowercase if it succeeds and NULL if an error occurs.

Examples

This expression returns Boston, Massachusetts:

```
WordCap("boston, MASSACHUSETTS")
```

This expression concatenates the characters in the emp_fname and emp_lname columns and makes the first letter of each word uppercase:

```
WordCap(emp_fname + " " + emp_lname)
```

Year

Description

Determines the year of a date value.

Syntax

Year (*date*)

Parameter	Description
date	The date value from which you want the year

Return value

Integer. Returns an integer whose value is a four-digit year adapted from the year portion of *date* if it succeeds and 1900 if an error occurs.

If the year is two digits, then PowerBuilder chooses the century, as follows. If the year is between 00 to 49, PowerBuilder assumes 20 as the first two digits; if it is between 50 and 99, PowerBuilder assumes 19.

Usage

Obtains the year portion of *date*. PowerBuilder handles years from 1000 to 3000 inclusive.

If your data includes date before 1950, such as birth dates, always specify a 4-digit year so that Year and other PowerBuilder functions, such as Sort, interpret the date as intended.

Example

This expression returns 1995:

```
Year(1995-01-31)
```

See also

Day
Month
Year in Chapter 1, "PowerScript Functions"

CHAPTER 3

Using Text Patterns

A text pattern is an expression that you can use to evaluate whether a string contains a particular pattern of characters. Text patterns consist of metacharacters, which have special meaning, and characters (non-metacharacters). The way the metacharacters and normal characters are combined specify the pattern you are looking for.

Text patterns are used with the Match function, which can be used both in scripts and DataWindow painter expressions.

This chapter explains the meanings of metacharacters and non-metacharacters in text patterns and provides examples of some common text patterns.

Metacharacters and non-metacharacters

Valid metacharacters are:

- ^ (caret)
- $ (dollar sign)
- . (period)
- \ (backslash)
- [ ] (brackets)
- * (asterisk)
- + (plus)
- ? (question mark)

The following tables explain the meaning and use of these metacharacters:

Metacharacter	Meaning	Example
Caret (^)	Matches the beginning of a string	^C matches C at the beginning of a string.
Dollar sign ($)	Matches the end of a string	s$ matches s at the end of a string.
Period (.)	Matches any character	. . . matches three consecutive characters.
Backslash (\)	Removes the following metacharacter's special characteristics so that it matches itself	\$ matches $.
Character class (a group of characters enclosed in square brackets ([ ]))	Matches any of the enclosed characters	[AEIOU] matches A, E, I, O, or U. You can use hyphens to abbreviate ranges of characters in a character class. For example, [A-Za-z] matches any letter.
Complemented character class (first character inside the brackets is a caret)	Matches any character *not* in the group following the caret	[^0-9] matches any character except a digit, and [^A-Za-z] matches any character except a letter.

The metacharacters asterisk (*), plus (+), and question mark (?) are unary operators that are used to specify repetitions in a regular expression:

Metacharacter	Meaning	Example
* (asterisk)	Indicates zero or more occurrences	A* matches zero or more As (no As, A, AA, AAA, and so on).
+ (plus)	Indicates one or more occurrences	A+ matches one A or more than one A (A, AAA, and so on).
? (question mark)	Indicates zero or one occurrence	A? matches an empty string ("") or A.

Non-metacharacters

You can use characters that are not metacharacters in a text pattern. One or more non-metacharacters in a text pattern (such as letters) match the characters themselves. For example, A matches A, and ABC matches ABC.

Text pattern examples

The following table shows various text patterns and sample text that matches each pattern:

This pattern	Matches
AB	Any string that contains AB; for example, ABA, DEABC, graphAB_one
B*	Any string that contains 0 or more Bs; for example, AC, B, BB, BBB, ABBBC, and so on
AB*C	Any string containing the pattern AC or ABC or ABBC, and so on (0 or more Bs)
AB+C	Any string containing the pattern ABC or ABBC or ABBBC, and so on (1 or more Bs)
ABB*C	Any string containing the pattern ABC or ABBC or ABBBC, and so on (1 B plus 0 or more Bs)
^AB	Any string starting with AB
AB?C	Any string containing the pattern AC or ABC (0 or 1 B)
^[ABC]	Any string starting with A, B, or C
[^ABC]	A string containing any characters other than A, B, or C
^[^abc]	A string that begins with any character except a, b, or c
^[^a-z]$	Any single-character string that is not a lowercase letter (^ and $ indicate the beginning and end of the string)
[A-Z]+	Any string with one or more uppercase letters
^[0-9]+$	Any string consisting only of digits
^[0-9][0-9][0-9]$	Any string consisting of exactly three digits
^([0-9][0-9][0-9])$	Any consisting of exactly three digits enclosed in parentheses

APPENDIX A

DataWindow Object Attributes

About this appendix

This appendix describes the attributes that control the appearance and behavior of a DataWindow. There are many ways you can affect these values during execution. There are several functions that get and set specific attributes, and you can use the general-purpose Describe and Modify functions to get and set attribute values. A subset of the attributes can be used with the SyntaxFromSQL function to generate DataWindow source code. You can use that code in the Create function to create new DataWindows. Appendix B lists the syntax for DataWindow source code.

The tables in the first part of this appendix list the attributes for each object within a DataWindow object, with short descriptions. There are also tables for SyntaxFromSQL object keywords. After the first table of DataWindow attributes, the tables are alphabetical by object and keyword name.

An annotation at the beginning of the table and checkmark columns identifies whether you can use that attribute with Describe (**D**), Modify (**M**), or SyntaxFromSQL (**S**). When (*exp*) is included in the description, then you can specify a DataWindow painter expression as the value for that attribute. A DataWindow painter expression lets you specify conditions for determining the attribute value.

The second half of this appendix is an alphabetical list of attributes with descriptions, syntax, and examples. When you find an attribute you want to use in the first part, look up the attribute in the alphabetical list to find the specific syntax you need to use. In the tables that describe the attribute values, (*exp*) again indicates that you can use a DataWindow painter expression for the value.

For more information and examples of setting attributes, see the Describe, Modify, and SyntaxFromSQL functions in Chapter 1.

Contents

Objects in a DataWindow and their attributes

The following tables for DataWindow objects and SyntaxFromSQL keywords list the attributes that apply to the objects and keywords. There are tables for:

Attributes for the DataWindow object

(Used with Describe, Modify, and SyntaxFromSQL)

Attribute for the DataWindow	D	M	S	Description
Attributes	x			All general attributes
Bands	x			List of bands
Bandname.attribute	x	x		Color, height, etc. for a band
Color	x	x	x	Text color
Column.Count	x			Number of columns
Crosstab.*attribute*	x	x		Settings for a crosstab DataWindow
Data	x			Description of data
Detail.*attribute*	x	x		See *Bandname.attribute*
FirstRowOnPage	x			The row number of the first displayed row
Font.Bias	x	x		Treat fonts as display or printer
Footer.*attribute*	x	x		See *Bandname.attribute*

Attribute for the DataWindow	D	M	S	Description
Grid.ColumnMove	x	x		Whether the user can drag to reposition columns
Grid.Lines	x	x		Options for lines in grid DataWindow
Header.#.*attribute*	x	x		See *Bandname.attribute*
Header.*attribute*	x	x		See *Bandname.attribute*
Help.*attribute*	x	x		Help settings for specific DataWindow actions
HorizontalScrollMaximum	x			Width of scroll box in the horizontal scrollbar
HorizontalScrollMaximum2	x			Width of second scroll box when scrollbar is split
HorizontalScrollPosition	x	x		The position of the scroll box in the scrollbar
HorizontalScrollPosition2	x	x		The position of scroll box in second split scrollbar
HorizontalScrollSplit	x	x		The position of the split in the scrollbar
Label.*attribute*	x	x	x	Various settings for the label presentation style
LastRowOnPage	x			The last visible row on the page
Message.Title	x	x	x	The title of the dialog box that displays errors
Nested	x			Whether the DataWindow has nested reports
Objects	x			The objects in the DataWindow
OLE.Client.*attribute*	x	x		Settings for the DataWindow as an OLE client
Pointer	x	x		(*exp*) The pointer when over the DataWindow
Print.*attribute*	x	x	x	Various settings for printing
Printer	x			The currently selected printer
Processing	x			The processing required by the presentation style
QueryMode	x	x		Whether the DataWindow is in query mode
QuerySort	x	x		Whether the result set from the query is sorted
ReadOnly	x	x		Whether the DataWindow is read-only
Retrieve.AsNeeded	x	x		Whether data will be retrieved only as needed
Row.Resize	x	x		Whether the user can change the height of rows
Rows_Per_Detail			x	Number of rows in each column of N-Up style
Selected	x	x		List of selected objects
Selected.Data	x			List of selected data
Selected.Mouse	x	x		Whether the user can use the mouse to select
ShowDefinition	x	x		(*exp*) Display column names instead of data
Sparse	x	x		(*exp*) The repeating columns to be suppressed
Storage	x			The amount of storage used by the DataWindow
Summary.*attribute*	x	x		See *Bandname.attribute*
Syntax	x			The syntax of the DataWindow
Syntax.Data	x			The data of the DataWindow in parse format
Syntax.Modified	x	x		Whether the syntax has been modified
Table.*attribute*	x	x		Various settings for the database
Timer_Interval	x	x	x	The milliseconds between timer events
Trailer.#.*attribute*	x	x		See *Bandname.attribute*
Units	x		x	The unit of measure for the DataWindow
VerticalScrollMaximum	x			The height of the scroll box in the scrollbar
VerticalScrollPosition	x	x		The position of the scroll box in the scrollbar
Zoom	x	x		The scaling percentage of the DataWindow

Attributes for bitmap objects

(Used with Describe and with Modify as marked)

Attribute for a bitmap	M	Description
Attributes		A list of the attributes of the bitmap
Band		The band containing the bitmap
Border	x	(*exp*) The type of border around the bitmap
Filename	x	(*exp*) The file containing the bitmap
Height	x	(*exp*) The height of the bitmap
Invert	x	(*exp*) Whether the colors are displayed inverted
Moveable	x	Whether the user can move the bitmap
Pointer	x	(*exp*) The pointer image when it is over the bitmap
Resizeable	x	Whether the user can resize the bitmap
SlideLeft	x	Whether the bitmap moves left to fill in empty space
SlideUp	x	How the bitmap moves up to fill in empty space
Tag	x	(*exp*) The tag text for the bitmap
Type		The object's type, which is bitmap
Visible	x	(*exp*) Whether the bitmap object is visible
Width	x	(*exp*) The width of the bitmap
X	x	(*exp*) The x coordinate of the bitmap
Y	x	(*exp*) The y coordinate of the bitmap

Attributes for column objects

(Used with Describe, Modify, and SyntaxFromSQL)

Attribute for a column	D	M	S	Description
Accelerator	x	x		(*exp*) The accelerator key for the column
Alignment	x	x		(*exp*) The alignment of the column's text
Attributes	x			A list of the attributes of the column
Background.*attribute*	x	x	x	(*exp*) Background settings for the column
Band	x			The band containing the column
BitmapName	x			Whether the column's contents names a bitmap that will be displayed instead of the text
Border	x	x	x	(*exp*) The type of border around the column
CheckBox.*attribute*	x	x		Settings for CheckBox edit style
Color	x	x	x	(*exp*) The text color
ColType	x			The column's data type
Criteria.*attribute*	x	x		Settings for column in the Prompt for Criteria dialog
dbName	x	x		The name of the database column
dddw.*attribute*	x	x		Settings for DropDownDataWindow edit style
ddlb.*attribute*	x	x		Settings for DropDownListBox edit style

Attribute for a column	D	M	S	Description
Edit.*attribute*	x	x	x	Settings for Edit edit style
EditMask.*attribute*	x	x		Settings for EditMask edit style
Font.*attribute*	x	x	x	(*exp*) Font settings for the column text
Format	x	x		(*exp*) The column's display format
Height	x	x		(*exp*) The height of the column
Height.AutoSize	x	x		Whether the column's height is adjusted to fit the data
ID	x			The number of the column
Initial	x	x		The initial value in the column for a new row
Key	x	x		Whether the column is part of the table's primary key
Moveable	x	x		Whether the user can move the column
Name	x			The name of the column
Pointer	x	x		(*exp*) The pointer image when it is over the column
Protect	x	x		(*exp*) Whether the column is protected from changes
RadioButton.*attribute*	x	x		Settings for RadioButton edit style
Resizeable	x	x		Whether the user can resize the column
SlideLeft	x	x		Whether the column moves left to fill in empty space
SlideUp	x	x		How the column moves up to fill in empty space
TabSequence	x	x		The position of the column in the tab order
Tag	x	x		(*exp*) The tag text for the column
Type	x			The object's type, which is column
Update	x	x		Whether the column is updateable
Validation	x	x		(*exp*) The validation expression for the column
ValidationMsg	x	x		(*exp*) The message displayed when validation fails
Values	x	x		The values in the column's code table
Visible	x	x		(*exp*) Whether the column object is visible
Width	x	x		(*exp*) The width of the column
X	x	x		(*exp*) The x coordinate of the column
Y	x	x		(*exp*) The y coordinate of the column

Attributes for computed field objects

(Used with Describe and with Modify as marked)

Attribute for a computed field	M	Description
Alignment	x	(*exp*) The alignment of the computed field's text
Attributes		A list of the attributes of the computed field
Background.*attribute*	x	(*exp*) Background settings for the computed field
Band		The band containing the computed field
Border	x	(*exp*) The type of border around the computed field
Color	x	(*exp*) The text color
Expression	x	(*exp*) The expression for the computed field
Font.*attribute*	x	(*exp*) Font settings for the computed field

Attribute for a computed field	M	Description
Format	x	(*exp*) The computed field's display format
Height	x	(*exp*) The height of the computed field
Height.AutoSize	x	Whether the computed field's height is adjusted to fit the data
Moveable	x	Whether the user can move the computed field
Pointer	x	(*exp*) The pointer image when it is over the computed field
Resizeable	x	Whether the user can resize the computed field
SlideLeft	x	Whether the computed field moves left to fill in empty space
SlideUp	x	How the computed field moves up to fill in empty space
Tag	x	(*exp*) The tag text for the computed field
Type		The object's type, which is Compute
Visible	x	(*exp*) Whether the computed field object is visible
Width	x	(*exp*) The width of the computed field
X	x	(*exp*) The x coordinate of the computed field
Y	x	(*exp*) The y coordinate of the computed field

Attributes for graph objects

(Used with Describe and with Modify as marked)

Attribute for a graph	M	Description
Attributes		A list of the attributes of the graph
Axis	x	(*exp*) List of items (categories, series, or values) for the axis
Axis.attribute	x	(*exp*) Attributes for a graph axis
BackColor	x	(*exp*) The background color of the graph
Band		The band containing the graph
Border	x	(*exp*) The type of border around the graph
Category	x	See *Axis* and *Axis.attribute*
Color	x	(*exp*) The text color
Depth	x	(*exp*) The depth of a 3D graph
DispAttr.*fontattribute*	x	Font settings for various components of the graph
Elevation	x	(*exp*) The elevation of a 3D graph
GraphType	x	(*exp*) The type of graph (pie, bar, etc.)
Height	x	(*exp*) The height of the graph
Legend	x	(*exp*) The location of the legend
Legend.DispAttr	x	(*exp*) Display attributes for the legend (See DispAttr)
Moveable	x	Whether the user can move the graph
OverlapPercent	x	(*exp*) The overlap between data markers in different series
Perspective	x	(*exp*) The distance of the graph from the front of the window
Pointer	x	(*exp*) The pointer image when it is over the graph
Range		The rows in the DataWindow that are included in the graph
Resizeable	x	Whether the user can resize the graph
Rotation	x	(*exp*) The left-to-right rotation of a 3D graph

Attribute for a graph	M	Description
Series	x	See *Axis* and *Axis.attribute*
ShadeColor	x	(*exp*) The color of the back edge for 3D data markers
SizeToDisplay	x	Whether the graph is automatically sized to the display area
SlideLeft	x	Whether the graph moves left to fill empty space
SlideUp	x	How the graph moves up to fill empty space
Spacing	x	(*exp*) The gap between categories
Tag	x	(*exp*) The tag text for the graph
Title	x	(*exp*) The graph's title
Title.DispAttr	x	(*exp*) Display attributes for the title (See DispAttr)
Type		The object's type, which is graph
Values	x	See *Axis* and *Axis.attribute*
Visible	x	(*exp*) Whether the graph object is visible
Width	x	(*exp*) The width of the graph
X	x	(*exp*) The x coordinate of the graph
Y	x	(*exp*) The y coordinate of the graph

Attributes for the Group keyword

(Used with SyntaxFromSQL)

Attribute	Description
NewPage	Whether a change in a group column's value causes a page break
ResetPageCount	Whether a new value in a group column restarts page numbering

Attributes for line objects

(Used with Describe and with Modify as marked)

Attribute for a line	M	Description
Attributes		A list of the attributes of the line
Background.*attribute*	x	(*exp*) Background settings for the line
Band		The band containing the line
Moveable	x	Whether the user can move the line
Pen.*attribute*	x	(*exp*) Appearance settings of the line
Pointer	x	(*exp*) The pointer image when it is over the line
Resizeable	x	Whether the user can resize the line
SlideLeft	x	Whether the line moves left to fill empty space
SlideUp	x	How the line moves up to fill empty space
Tag	x	(*exp*) The tag text for the line
Type		The object's type, which is line

Attribute for a line	M	Description
Visible	x	(*exp*) Whether the line object is visible
X1	x	(*exp*) The x coordinate of one end of the line
X2	x	(*exp*) The x coordinate of the other end of the line
Y1	x	(*exp*) The y coordinate of one end of the line
Y2	x	(*exp*) The y coordinate of the other end of the line

Attributes for oval, rectangle, and roundrectangle objects

(Used with Describe and with Modify as marked)

Attribute	M	Description
Attributes		A list of the attributes of the object
Background.*attribute*	x	(*exp*) Background settings for the object
Band		The band containing the object
Brush.*attribute*	x	(*exp*) Settings for fill pattern and color
Height	x	(*exp*) The height of the object
Moveable	x	Whether the user can move the object
Pen.*attribute*	x	(*exp*) Appearance settings of the object
Pointer	x	(*exp*) The pointer image when it is over the object
Resizeable	x	Whether the user can resize the object
SlideLeft	x	Whether the object moves left to fill empty space
SlideUp	x	How the object moves up to fill empty space
Tag	x	(*exp*) The tag text for the object
Type		The object's type, which is ellipse, rectangle, or roundrectangle
Visible	x	(*exp*) Whether the object is visible
Width	x	(*exp*) The width of the object
X	x	(*exp*) The x coordinate of the object
Y	x	(*exp*) The y coordinate of the object

Additional attributes for roundrectangle objects

(Used with Describe and with Modify as marked)

Attribute	M	Description
EllipseHeight	x	(*exp*) The radius of the vertical part of the rounded corner
EllipseWidth	x	(*exp*) The radius of the horizontal part of the rounded corner

See also Oval and Rectangle

Attributes for report objects

(Used with Describe and with Modify as marked)

Attribute for a report	M	Description
Attributes		A list of the attributes of the report
Band		The band containing the report
Border	x	(*exp*) The type of border around the report
Criteria	x	The WHERE clause that relates the report to the main DataWindow
DataObject	x	The name of the DataWindow that is the nested report
Height	x	(*exp*) The height of the report
Moveable	x	Whether the user can move the report
Nest_Arguments	x	Retrieval arguments for the report
NewPage	x	Whether to start the report on a new page (composite only)
Pointer	x	(*exp*) The pointer image when it is over the report
Resizeable	x	Whether the user can resize the report
SlideLeft	x	Whether the report moves left to fill in empty space
SlideUp	x	How the report moves up to fill in empty space
Tag	x	(*exp*) The tag text for the report
Trail_Footer	x	Where to print the footer (composite only)
Type		The object's type, which is report
Visible	x	(*exp*) Whether the report object is visible
Width	x	(*exp*) The width of the report
X	x	(*exp*) The x coordinate of the report
Y	x	(*exp*) The y coordinate of the report

Attributes for the Style keyword

(Used with SyntaxFromSQL)

Attribute	Description
Detail_Bottom_Margin	Bottom margin of the detail area
Detail_Top_Margin	Top margin of the detail area
Header_Bottom_Margin	Bottom margin of the header area
Header_Top_Margin	Top margin of the header area
Horizontal_Spread	Horizontal space between columns in the detail area
Left_Margin	The left margin of the DataWindow
Report	Whether the DataWindow is a read-only report
Type	The presentation style
Vertical_Size	The height of the columns in the detail area
Vertical_Spread	The vertical space between columns in the detail area

Attributes for TableBlob objects

(Used with Describe and with Modify as marked)

Attribute for a TableBlob	M	Description
Attributes		A list of the attributes of the TableBlob
Band		The band containing the TableBlob
Border	x	(*exp*) The type of border around the TableBlob
ClientName	x	(*exp*) The name of the OLE client in the server window
Height	x	(*exp*) The height of the TableBlob
ID		The number of the TableBlob
KeyClause	x	(*exp*) The key clause used when retrieving the blob
Moveable	x	Whether the user can move the TableBlob
OLEClass	x	(*exp*) The name of the TableBlob's OLE column
Pointer	x	(*exp*) The pointer image when it is over the TableBlob
Resizeable	x	Whether the user can resize the TableBlob
SlideLeft	x	Whether the TableBlob moves left to fill empty space
SlideUp	x	How the TableBlob moves up to fill empty space
Table	x	(*exp*) The database table that contains the blob
Tag	x	(*exp*) The tag text for the object
Template	x	(*exp*) The file used to start the OLE application
Type		The object's type, which is tableblob
Visible	x	(*exp*) Whether the TableBlob is visible
Width	x	(*exp*) The width of the TableBlob
X	x	(*exp*) The x coordinate of the TableBlob
Y	x	(*exp*) The y coordinate of the TableBlob

Attributes for text objects

(Used with Describe and with Modify and SyntaxFromSQL as marked)

Attribute for text	M	S	Description
Alignment	x	x	The alignment of the text
Attributes			A list of the attributes of the text object
Background.*attribute*	x	x	(*exp*) Background settings for the text object
Band			The band containing the text object
Border	x	x	(*exp*) The type of border around the text object
Color	x	x	(*exp*) The text color
Font.*attribute*	x	x	(*exp*) Font settings for the text
Height	x		(*exp*) The height of the text object
Height.AutoSize	x		Whether the object's height is adjusted to fit the data
Moveable	x		Whether the user can move the text object
Pointer	x		(*exp*) The pointer image when it is over the text object
Resizeable	x		Whether the user can resize the text object

Attribute for text	M	S	Description
SlideLeft	x		Whether the text object moves left to fill empty space
SlideUp	x		How the text object moves up to fill empty space
Tag	x		(*exp*) The tag text for the text object
Text	x		(*exp*) The displayed text
Type			The object's type, which is text
Visible	x		(*exp*) Whether the object is visible
Width	x		(*exp*) The width of the text object
X	x		(*exp*) The x coordinate of the text object
Y	x		(*exp*) The y coordinate of the text object

Title keyword

(Used with SyntaxFromSQL)

Attribute	Description
Title("string")	The title for the DataWindow

Alphabetical list of attributes

Accelerator

Description

The accelerator key that a user can press to select a column in the DataWindow object. The column does not display the key but you can include an underlined letter in the text object that labels the column.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.Accelerator{='*acceleratorkey* '}"

Parameter	Description
columnname	The name of the column for which you want to get or set the accelerator key.
acceleratorkey	(*exp*) A string expression whose value is the letter that will be the accelerator key for *columnname*. *Acceleratorkey* can be a quoted DataWindow painter expression.

Examples

```
ls_data = dw_1.Describe("emp_name.Accelerator")
dw_1.Modify("emp_name.Accelerator='A'")
```

Alignment

Description

The alignment of the object's text within its borders.

Applies to

Column, Computed Field, Text objects

Used in

Describe and Modify (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax

Describe and Modify:

"*objectname*.Alignment { ='*alignmentvalue* ' }"

SyntaxFromSQL:

Text(... Alignment=*value* ...)

Parameter	Description
objectname	The name of the object for which you want to get or set the alignment.

Parameter	Description
alignmentvalue	(*exp*) A number specifying the type of alignment for the text of *objectname*. *Alignmentvalue* can be a quoted DataWindow painter expression. Values are: ♦ 0 — (Default) Left ♦ 1 — Right ♦ 2 — Center When generating DataWindow syntax with SyntaxFromSQL, the setting for Alignment applies to all text objects user as column labels.

Examples

```
ls_data = dw_1.Describe("emp_name.Alignment")
dw_1.Modify("emp_name_t.Alignment='2'")
```

Arguments

Description

The retrieval arguments required by the data source. You specify retrieval arguments in the DataWindow's SELECT statement and you provide values for the retrieval arguments when you call the Retrieve function.

Applies to

Database table for the DataWindow object

Used in

Not settable in PowerScript. Shown in DataWindow syntax (Appendix B).

Syntax

Table(Arguments=((*name1*, *type*), (*name2*, *type*). . .) . . .)

Parameter	Description
name	The name of the retrieval argument
type	The type of the argument: ♦ Date or a Date list ♦ DateTime or a DateTime list ♦ Number or a Number list ♦ String or a String list ♦ Time or a Time list

Attributes

Description: A tab-separated list of all the attributes that apply to an object.

Applies to: DataWindow, Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in: Describe

Syntax: "*objectname*.Attributes"

Examples:

```
ls_data = dw_1.Describe("DataWindow.Attributes")
ls_data = dw_1.Describe("emp_name_t.Attributes")
```

Axis

Description: The list of items associated with an axis of a graph. Each item is separated by a comma. You can ask for the list of categories on the Category axis, the series on the Series axis, or the values on the Values axis.

Applies to: Graph objects

Used in: Describe, Modify

Syntax: "*graphname.axis* { ='*list*' }"

Parameter	Description
graphname	The name of the graph within the DataWindow object for which you want to get or set the list of items for *axis*.
axis	An axis name. Values are: ♦ Category ♦ Series ♦ Values
list	A string listing the categories, series, or values for the graph. The contents of the list depends on the axis you specify. The items in the list are separated by commas. *List* is quoted.

Examples:

```
ls_data = dw_1.Describe("gr_1.Category")
ls_data = dw_1.Describe("gr_1.Series")
ls_data = dw_1.Describe("gr_1.Values")
dw_1.Modify("gr_1.Series='Actual, Budget'")
```

Axis.attribute

Description: Settings that control the appearance of an axis on a graph.

Applies to: Graph objects

Used in: Describe, Modify

Syntax: "*graphname.axis.attribute* { =*value* }"

Parameter	Description
graphname	The name of the graph within the DataWindow object for which you want to get or set an attribute value for an axis.
axis	An axis name. Values are: ♦ Category ♦ Series ♦ Values
attribute	An attribute for the axis. Attributes and their settings are listed in the following table.
value	The value to be assigned to the attribute when you use Modify. For axis attributes, *value* can be a quoted DataWindow painter expression.

Attribute for *Axis*	Value
AutoScale	(*exp*) A boolean number specifying whether PowerBuilder scales the axis automatically. Values are: ♦ 0 — No, do not automatically scale the axis ♦ 1 — Yes, automatically scale the axis
DispAttr.-*fontattribute*	(*exp*) Attributes that control the appearance of the text that labels the axis divisions. See DispAttr.*fontattribute*.
DisplayEvery-NLabels	(*exp*) An integer specifying which major axis divisions to label. For example, 2 means label every other tick mark. If the labels are too long, they are clipped. 0 means to label as often as possible without clipping.

Attribute for *Axis*	Value
DropLines	(*exp*) An integer indicating the type of drop line for the axis. Values are: ♦ 0 — None ♦ 1 — Solid ♦ 2 — Dash ♦ 3 — Dot ♦ 4 — DashDot ♦ 5 — DashDotDot
Frame	(*exp*) An integer indicating the type of line used for the frame. Values are 0–5. See DropLines in this table for their meaning.
Label	(*exp*) A string whose value is the axis label.
LabelDispAttr.-*fontattribute*	(*exp*) Attributes that control the appearance of the axis label. See DispAttr.*fontattribute*.
MajorDivisions	(*exp*) An integer specifying the number of major divisions on the axis.
MajorGridLine	(*exp*) An integer specifying the type of line for the major grid. Values are 0–5. See DropLines in this table for their meaning.
MajorTic	(*exp*) An integer specifying the type of the major tick marks. Values are: ♦ 1 — None ♦ 2 — Inside ♦ 3 — Outside ♦ 4 — Straddle
MaximumValue	(*exp*) A double specifying the maximum value for the axis.
MinimumValue	(*exp*) A double specifying the minimum value for the axis.
MinorDivisions	(*exp*) An integer specifying the number of minor divisions on the axis.
MinorGridLine	(*exp*) An integer specifying the type of line for the minor grid. Values are 0–5. See DropLines in this table for their meaning.

Attribute for *Axis*	Value
MinorTic	(*exp*) An integer specifying the type of the minor tick marks. Values are: ♦ 1 — None ♦ 2 — Inside ♦ 3 — Outside ♦ 4 — Straddle
OriginLine	(*exp*) An integer specifying the type of origin line for the axis. Values are 0–5. See DropLines in this table for their meaning.
PrimaryLine	(*exp*) An integer specifying the type of primary line for the axis. Values are 0–5. See DropLines in this table for their meaning.
RoundTo	(*exp*) A double specifying the value to which you want to round the axis values.
RoundToUnit	(*exp*) An integer specifying the units for the rounding value. The units must be appropriate for the axis data type. Values are: ♦ 0 — Default, for an axis of any data type ♦ 1 — Years, for an axis of type date or DateTime ♦ 2 — Months, for an axis of type date or DateTime ♦ 3 — Days, for an axis of type date or DateTime ♦ 4 — Hours, for an axis of type time or DateTime ♦ 5 — Minutes, for an axis of type time or DateTime ♦ 6 — Seconds, for an axis of type time or DateTime ♦ 7 — Microseconds, for an axis of type time or DateTime
ScaleType	(*exp*) An integer specifying the type of scale used for the axis. Values are: ♦ 1 — Scale_Linear ♦ 2 — Scale_Log10 ♦ 3 — Scale_Loge
ScaleValue	(*exp*) An integer specifying the scale of values on the axis. Values are: ♦ 1 — Scale_Actual ♦ 2 — Scale_Cumulative ♦ 3 — Scale_Percentage ♦ 4 — Scale_CumPercent

Attribute for *Axis*	Value
SecondaryLine	(*exp*) An integer specifying the type of secondary line for the axis. Values are 0–5. See DropLines in this table for their meaning.
ShadeBackEdge	(*exp*) A boolean number specifying whether the back edge of the axis is shaded. Values are: ♦ 0 — No, the back edge is not shaded ♦ 1 — Yes, the back edge is shaded
Sort	(*exp*) An integer specifying the way the axis values should be sorted. (Does not apply to the Values axis.) Values are: ♦ 0 — Unsorted ♦ 1 — Ascending ♦ 2 — Descending

Examples

```
ls_data = dw_1.Describe("gr_1.Category")
dw_1.Modify("gr_1.Series.AutoScale=0")
dw_1.Modify("gr_1.Values.Label='Cities'")
dw_1.Modify("gr_1.Category.LabelDispAttr.Alignment=2")
```

BackColor

Description

The background color of a graph in a DataWindow.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.BackColor { = *long* }"

Parameter	Description
graphname	The graph whose background color you want to get or set.
long	(*exp*) A long expression specifying the color (red, green, and blue values) to be used as the graph's background color. *Long* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.BackColor")
dw_1.Modify("graph_1.BackColor=250")
```

Background.*attribute*

Description

Settings for the color and transparency of an object.

Applies to

Column, Computed Field, Line, Oval, Rectangle, RoundRectangle, and Text objects

Used in

Describe and Modify (1st syntax), SyntaxFromSQL (2nd and 3rd syntax)

Syntax

Describe and Modify:

"*objectname*.Background.*attribute* { ='*value* ' }"

SyntaxFromSQL:

Column(Background.*attribute*=*value*)

Text(Background.*attribute*=*value*)

Parameter	Description
objectname	The object whose Background attributes you want to get or set. When generating DataWindow syntax with SyntaxFromSQL, the Background settings apply to all columns or all text objects.
attribute	An attribute that applies to the background of an object, as listed in the Attribute table below.
value	Values for the attributes are shown below. *Value* can be a quoted DataWindow painter expression.

Attribute for Background	Value
Color	(*exp*) A long expression specifying the color (the red, green, and blue values) to be used as the object's background color.
Mode	(*exp*) A number expression specifying the mode of the background of *objectname*. Values are: ♦ 0 — Make the object's background opaque ♦ 1 — Make the object's background transparent

Background color of a line
The background color of a line is the color that displays between the segments of the line when the pen style is not solid.

Transparent background
If Background.Mode is transparent (1), Background.Color is ignored.

Examples

```
ls_data = dw_1.Describe("oval_1.Background.Color")

dw_1.Modify("emp_name.Background.Color='11665407'")

ls_data = dw_1.Describe("emp_name.Background.Mode")

dw_1.Modify("emp_name.Background.Mode='1'")

dw_1.Modify("rndrect_1.Background.Mode='0'")

SQLCA.SyntaxFromSQL(sql_syntax,  &
   "Style(...) Column(Background.Mode=1 ...) ...",  &
   ls_Errors)

SQLCA.SyntaxFromSQL(sql_syntax,  &
   "Style(...) Column(Background.Color=11665407 ...)", &
   ls_Errors)
```

Band

Description

The band or layer in the DataWindow object that contains the object. The returned text is one of the following, where # is the level number of a group: detail, footer, header, header.#, summary, trailer.#, foreground, background.

Changing an object's band
Use the SetPosition function to change an object's band.

Applies to

Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, and Text objects

Used in

Describe

Syntax

"*objectname*.Band"

Parameter	Description
objectname	The name of the object within the DataWindow for which you want the band it occupies.

Example

```
ls_data = dw_1.Describe("emp_title.Band")
```

Bandname.attribute

Description

Settings for the color, size, and pointer of a band in the DataWindow object.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.*bandname*{.#}.*attribute* { =*value* }"

Parameter	Description
bandname	The identifier of a band in the DataWindow object. Values are: ♦ Detail ♦ Header ♦ Footer ♦ Summary ♦ Trailer
#	The number of the group you want when *bandname* is Header or Trailer. The group must exist.
attribute	An attribute that applies to the band, as listed in the Attribute table.
value	Values for the attributes are shown in the following table.

Attribute for *Bandname*	Value
Color	(*exp*) A long specifying the color (the red, green, and blue values) to be used as the band's background color. *Value* can be a quoted DataWindow painter expression.
Height	An integer specifying the height of the detail area in the unit of measure specified for the DataWindow.

Attribute for *Bandname*	Value
Height.AutoSize	(Only when *bandname* is Detail) Whether the height of the band in the DataWindow object should be calculated by looking at all the objects that are autosized. Values are: ♦ No — Make all the heights the same ♦ Yes — Calculate the height according to the formula: Height = height of the largest painter row + painted row height – row height The height of the detail band will never be less than the height selected in the painter.
Pointer	(*exp*) A string specifying a value of the Pointer enumerated data type or the name of a cursor file (.CUR) to be used for the pointer. See the SetPointer function for a list of Pointer values. *Pointername* can be a quoted DataWindow painter expression.

Examples

```
ls_data = dw_1.Describe("DataWindow.Detail.Height")

ls_data =  &
   dw_1.Describe("DataWindow.Detail.Height.AutoSize")

dw_1.Modify("DataWindow.Detail.Pointer='hand.cur'")

dw_1.Modify("DataWindow.Detail.Pointer=' ~"Cross!~" ~t &
if(emp_status=~"a~", ~"HourGlass!~", ~"Cross!~")'")

dw_1.Modify("DataWindow.Footer.Height=250")

ll_color = RGB(200, 200, 500)
dw_1.Modify("DataWindow.Header.2.Color="  &
   + String(ll_color))

dw_1.Modify("DataWindow.Trailer.2.Height=500")

dw_1.Modify(  &
      "DataWindow.Summary.Pointer='c:\pb\total.cur'")
```

Bands

Description

A list of the bands in the DataWindow object. The list can include one or more of the following band identifiers, where # is the level number of a group: Detail, Footer, Header, Header.#, Summary, Trailer.#. The items in the list are separated by tabs.

Applies to

DataWindow

Used in

Describe

Syntax "DataWindow.Bands"

Example
```
ls_data = dw_1.Describe("DataWindow.Bands")
```

BitmapName

Description Whether PowerBuilder interprets the column's value as the name of a bitmap file and displays the bitmap instead of the text. BitmapName's value is either Yes or No.

Applies to Column objects

Used in Describe

Syntax "*columnname*.BitmapName"

Example
```
ls_data = dw_1.Describe("emp_name.BitmapName")
```

Border

Description The type of border for the object.

Applies to Bitmap, Column, Computed Field, Graph, Report, TableBlob, Text objects

Used in Describe and Modify (1st syntax), SyntaxFromSQL (2nd and 3rd syntax)

Syntax Describe and Modify:

"*objectname*.Border { ='*integer*' }"

SyntaxFromSQL:

Column(... Border=*value* ...)

Text(... Border=*value* ...)

Parameter	Description
objectname	The name of the object whose border you want to get or set. When generating DataWindow syntax with SyntaxFromSQL, the Border setting applies to all columns or all text objects.

Parameter	Description
integer	(*exp*) A number specifying the type of border. Values are: ♦ 0 — None ♦ 1 — Shadow ♦ 2 — Rectangle ♦ 3 — Resize ♦ 4 — Line ♦ 5 — 3D Lowered ♦ 6 — 3D Raised *Integer* can be a DataWindow quoted painter expression.

Examples

```
ls_data = dw_1.Describe("emp_name_t.Border")

dw_1.Modify("emp_name_t.Border='6'")

SQLCA.SyntaxFromSQL(sql_syntax,  &
   "Style(...) Column(Border=5 ...) ...", ls_Errors)
```

Brush.*attribute*

Description

Settings for the fill pattern and color of a graphic object.

Applies to

Oval, Rectangle, and RoundRectangle objects

Used in

Describe, Modify

Syntax

"*objectname*.Brush.*attribute* { ='*value* ' }"

Parameter	Description
objectname	The name of the line, oval, rectangle, round rectangle, or text object whose Brush attribute you want to get or set.
attribute	An attribute that applies to the Brush characteristics of an object, as listed in the Attribute table below.
value	Values for the attributes are shown below. *Value* can be a quoted DataWindow painter expression.

Attribute for Brush	Value
Color	(*exp*) A long expression specifying the color (the red, green, and blue values) to be used to fill the object.

Attribute for Brush	Value
Hatch	(*exp*) A number expression specifying the fill pattern of *objectname*. Values are: ♦ 0 — Horizontal ♦ 1 — Bdiagonal (lines from lower left to upper right) ♦ 2 — Vertical ♦ 3 — Cross ♦ 4 — Fdiagonal (lines from upper left to lower right) ♦ 5 — DiagCross ♦ 6 — Solid ♦ 7 — Transparent

Examples

```
ls_data = dw_1.Describe("oval_1.Brush.Hatch")
dw_1.Modify("oval_1.Brush.Hatch='5'")
dw_1.Modify("oval_1.Brush.Color='16731766'")
```

Category

See *Axis* and *Axis.attribute*.

CheckBox

Description: Settings for a column whose edit style is CheckBox.

Applies to: Column objects

Used in: Describe, Modify

Syntax: "*columnname*.CheckBox.*attribute* { =*value* }"

Parameter	Description
columnname	The column whose edit style is CheckBox for which you want to get or set attribute values.
attribute	An attribute for the CheckBox edit style, as listed in the Attribute table below.
value	Values for the attributes are shown in the Attribute table. For CheckBox attributes, *value* cannot be a DataWindow painter expression.

Attribute for CheckBox	Value
3D	Whether the CheckBox should be 3D. Values are: ♦ Yes — Make the CheckBox 3D ♦ No — Do not make the CheckBox 3D
LeftText	Whether the CheckBox label is to the left or right of the CheckBox. Values are: ♦ Yes — Display the label on the left ♦ No — Display the label on the right
Off	A string constant specifying the column value when the CheckBox is off (unchecked). The value must be the same data type as the column.
On	A string constant specifying the value that will be put in the column when the CheckBox is on (checked). The value must be the same data type as the column.
Other	A string constant specifying the value that will be put in the column when the CheckBox is in the third state (neither checked nor unchecked). The value must be the same data type as the column.
Scale	Whether you want to scale the two-dimensional CheckBox. Takes effect only when the 3D attribute is No. Values are: ♦ Yes — Scale the CheckBox ♦ No — Do not scale the CheckBox
Text	A string specifying the CheckBox's label text.

Examples

```
dw_1.Modify("emp_gender.CheckBox.3D=no")

IF dw_1.Describe("emp_status.CheckBox.LeftText")  &
      = "yes" THEN
   dw_1.Modify("emp_status2.CheckBox.LeftText=yes")
END IF

dw_1.Modify("emp_status.CheckBox.Off='Terminated'")

dw_1.Modify("emp_status.CheckBox.On='Active'")

dw_1.Modify("emp_status.CheckBox.Other='Unknown'")
```

ClientName

Description

The name of the OLE client. The default is "Untitled." ClientName is used by some applications in the server window's title.

Applies to

TableBlob objects

Used in

Describe, Modify

Syntax

"*tblobname*.ClientName { = '*clientname* ' }"

Parameter	Description
tblobname	The name of a blob column in the DataWindow object.
clientname	(*exp*) A string expression to be used in the title of the server application's window. The string usually includes data from the current row so that the window title can identify the blob's row. *Clientname* is quoted and can be a DataWindow painter expression.

Examples

```
cname = dw_1.Describe("emppict_blob.ClientName")

dw_1.Modify("emppict_blob.ClientName='" +  &
    "~"Data for ~" + String(emp_id)'")
```

Color

Description

The text color of the column or the background color of the DataWindow.

Applies to

DataWindow or Column objects

Used in

Describe and Modify (1st and 2nd syntax),
SyntaxFromSQL (3rd and 4th syntax)

Syntax

Describe and Modify:

"DataWindow.Color { = long }"

"columnname.Color { = long }"

SyntaxFromSQL:

DataWindow(Color=long)

Column(Color=long)

Parameter	Description
columnname	The column whose text color you want to set
long	(*exp* for columns only) A long value specifying the color of the column text or the DataWindow background. When you are specifying the text color of a column, you can specify a DataWindow painter expression in quotes. You cannot specify an expression for the DataWindow background color. When generating DataWindow syntax with SyntaxFromSQL, the Color setting for Column applies to all columns.

See also Background, BackColor

ColType

Description The data type of the column.

Applies to Column objects

Used in Describe

Syntax "*columnname*.ColType"

Parameter	Description
columnname	The column for which you want the data type. Possible data types are: ♦ Char (*n*) — *n* is the number of characters ♦ Date ♦ DateTime ♦ Decimal (*n*) — *n* is the number of decimal places ♦ Number ♦ Time ♦ Timestamp

Example

```
ls_coltype = dw_1.Describe("emp_id.ColType")
```

Column.Count

Description The number of columns in the DataWindow object.

Applies to DataWindow

Used in Describe

Syntax "DataWindow.Column.Count"

Example
```
ls_colcount = dw_1.Describe("DataWindow.Column.Count")
```

Criteria

Description The WHERE clause for a related report. The Criteria attribute defines the connection between the related report and the DataWindow.

Applies to Report objects

Used in Describe, Modify

Syntax "*reportname*.Criteria { = *string* }"

Parameter	Description
reportname	The name of the report object for which you want to get or set Criteria
string	An expression that will be the WHERE clause for the related report

Example
```
ls_colcount = dw_1.Describe("rpt_1.Criteria")
dw_1.Modify("rpt_1.Criteria='emp_id=:emp_id'")
```

Criteria.*attribute*

Description Settings for the Prompt for Criteria dialog. When Prompt for Criteria is enabled, PowerBuilder prompts the user to specify criteria for retrieving data whenever the Retrieve function is called. Note that the Required attribute also affects query mode.

Applies to Column objects

Used in Describe, Modify

Syntax "*columnname*.Criteria.*attribute* { = *value* }"

Parameter	Description
columnname	The name of the column for which you want to get or set Prompt for Criteria attributes.

Parameter	Description
attribute	An attribute for the Prompt for Criteria dialog. Attributes and their settings are listed in the table below.
value	A Yes or No value to be assigned to the attribute when you use Modify. For Criteria attributes, *value* cannot be a DataWindow painter expression.

Attribute for Criteria	Value
Dialog	Whether Prompt for Criteria is on for *columnname*. Values are: ♦ Yes — Include *columnname* in the Prompt for Criteria dialog ♦ No — (Default) Do not include *columnname* in the Prompt for Criteria dialog If the Dialog attribute is Yes for at least one column in the DataWindow, then PowerBuilder displays the Prompt for Criteria dialog when the Retrieve function is called.
Override_Edit	Whether the user must enter data in the Prompt for Criteria dialog according to the edit style defined for the column in the DataWindow object or be allowed to enter any specifications in a standard edit box. Values are: ♦ Yes — Allow the user to override the column's edit style and enter data in a standard edit box ♦ No — (Default) Constrain the user to the edit style for the column
Required	Whether the user is restricted to equality and inequality operators (= and <>) when specifying criteria in query mode and in the Prompt for Criteria dialog. Values are: ♦ Yes — Require the user to use equality and inequality operators only ♦ No — (Default) Allow the user to use any relational operator, including =, <>, <, >, >=, and <=

Examples

```
setting = dw_1.Describe("empname.Criteria.Dialog")
dw_1.Modify("empname.Criteria.Dialog=Yes")
dw_1.Modify("empname.Criteria.Override_Edit=Yes")
dw_1.Modify("empname.Criteria.Required=No")
```

```
IF dw_1.Describe("empname.Criteria.Edit.Style")  &
        = "dddw" THEN
   dw_1.Modify("empname.Criteria.Override_Edit=Yes")
END IF
```

Crosstab.*attribute*

Description

Settings for a DataWindow object whose presentation style is Crosstab.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Crosstab.*attribute* { =*value* }"

Parameter	Description
attribute	An attribute for a Crosstab DataWindow. Attributes and their settings are listed in the table below.
value	A string expression listing the items to be assigned to the attribute when you use Modify. For Crosstab attributes, *value* is always quoted and can be a DataWindow painter expression.

Attribute for Crosstab	Value
Columns	(*exp*) A string containing a comma- or tab-separated list of the names of columns that make up the columns of the crosstab. These are the columns that display across the top of the crosstab.
Rows	(*exp*) A string containing a comma- or tab-separated list of the names of columns that make up the rows of the crosstab.
SourceNames	(*exp*) A string containing a comma-separated list of column names to be displayed in the Crosstab Definition dialog. The default names are the column names from the database.
Values	(*exp*) A string containing a comma- or tab-separated list of expressions that will be used to calculate the values of the crosstab.

Examples

```
setting = dw_1.Describe("DataWindow.Crosstab.Columns")
dw_1.Modify("DataWindow.Crosstab.Columns='dept_id'")
dw_1.Modify("DataWindow.Crosstab.Rows='salary'")
```

```
dw_1.Modify("DataWindow.Crosstab.SourceNames="  &
   + "'Order Number, Item Number, Price'")

dw_1.Modify("DataWindow.Crosstab.Values='empname'")
```

Data

Description: A tab-separated list describing the data in the DataWindow object.

Applies to: DataWindow

Used in: Describe

Syntax: "DataWindow.Data"

Example:

```
setting = dw_1.Describe("DataWindow.Data")
```

DataObject

Description: The name of the DataWindow that is the nested report within the main DataWindow.

Applies to: Report objects

Used in: Describe, Modify

Syntax: "*reportname*.DataObject='*dwname*'"

Parameter	Description
reportname	The name of the report object in the main DataWindow for which you want to get or set the nested DataWindow
dwname	A string naming a DataWindow object in the application's libraries that is the DataWindow for the report within the main DataWindow

Example:

```
setting = dw_1.Describe("rpt_1.DataObject")

dw_1.Modify("rpt_1.DataObject='d_empdata'")
```

dbName

Description: The name of the database column. PowerBuilder uses this value to construct the update syntax.

Applies to: Column objects

Used in

Describe, Modify

Syntax

"*columnname*.dbName { = '*dbcolumnname*' }"

Parameter	Description
columnname	The name of the column for which you want the name of the corresponding database column
dbcolumnname	The name of the database column associated with *columnname*

Examples

```
dbcol = dw_1.Describe("emp_id.dbName")
dw_1.Modify("emp_id.dbName='emp_id'")
```

dddw.*attribute*

Description

Attributes that control the appearance and behavior of a column with the DropDownDataWindow edit style.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.dddw.*attribute* { =*value* }"

Parameter	Description
columnname	The name of a column that has the DropDownDataWindow edit style.
attribute	An attribute for the DropDownDataWindow column. Attributes and their settings are listed in the table below.
value	The value to be assigned to the attribute when you use Modify. For dddw attributes, *value* cannot be a DataWindow painter expression.

Attribute	Value
AllowEdit	Whether the user can type a value as well as choose from the DropDownDataWindow's list. Values are: ♦ Yes — Typing is allowed ♦ No — (Default) Typing is not allowed

Attribute	Value
AutoHScroll	Whether the DropDownDataWindow automatically scrolls horizontally when the user enters or deletes data. Values are: ♦ Yes — (Default) Scroll horizontally automatically ♦ No — Do not scroll automatically
Case	The case of the text in the DropDownDataWindow. Values are: ♦ Any — Character of any case allowed ♦ Upper — Characters converted to uppercase ♦ Lower — Characters converted to lowercase
DataColumn	A string whose value is the name of the data column in the associated DropDownDataWindow. *Value* is quoted.
DisplayColumn	A string whose value is the name of the display column in the associated DropDownDataWindow. *Value* is quoted.
HScrollBar	Whether a horizontal scrollbar displays in the DropDownDataWindow. Values are: ♦ Yes — Display a horizontal scrollbar ♦ No — Do not display a horizontal scrollbar
HSplitScroll	Whether the horizontal scrollbar is split. The user can adjust the split position. Values are: ♦ Yes — Split the horizontal scrollbar so the user can scroll the display and data columns separately ♦ No — The horizontal scrollbar is not split
Limit	An integer from 0–32767 specifying the maximum number of characters that can be entered in the DropDownDataWindow. Zero means unlimited.
Name	A string whose value is the name of the predefined DropDownDataWindow style associated with the column. Named styles are defined in the Database painter and can be reused. Specifying a name that has not been previously defined associates the name with the column but does not define a new edit style.
NilisNull	Whether to set the data value of the DropDownDataWindow to NULL when the user leaves the edit box blank. Values are: ♦ Yes — Make the Empty string NULL ♦ No — Do not make the empty string NULL
PercentWidth	An integer specifying the width of the dropdown portion of the DropDownDataWindow as a percentage of the column's width.

Attribute	Value
Required	Whether the column is required. Values are: ♦ Yes — Required ♦ No — (Default) Not required
ShowList	Whether the ListBox portion of the DropDownDataWindow displays when the column has focus. A down arrow does not display at the right end of the DropDownDataWindow when dddw.ShowList is yes. Values are: ♦ Yes —Display the list whenever the column has the focus ♦ No — Do not display the list until the user selects the column
UseAsBorder	Whether a down arrow displays at the right end of the DropDownDataWindow. Values are: ♦ Yes — Display the arrow ♦ No — Do not display the arrow Note that if ShowList is set to Yes, the column ignores the UseAsBorder attribute and the arrow never displays.
VScrollBar	Whether a vertical scrollbar displays in the DropDownDataWindow for long lists. Values are: ♦ Yes — Display a vertical scrollbar ♦ No — Do not display a vertical scrollbar

Examples

```
ls_data = dw_1.Describe("emp_status.dddw.AllowEdit")
dw_1.Modify("emp_status.dddw.Case=Any")
dw_1.Modify("emp_status.dddw.DataColumn='status_id'")
dw_1.Modify("emp_status.dddw.Limit=30")
dw_1.Modify("emp_status.dddw.Name='d_status'")
dw_1.Modify("emp_status.dddw.PercentWidth=120")
```

ddlb.*attribute*

Description — Attributes that control the appearance and behavior of a column with the DropDownListBox edit style.

Applies to — Column objects

Used in — Describe, Modify

Syntax

"*columnname*.ddlb.*attribute* { =*value* }"

Parameter	Description
columnname	The name of a column that has the DropDownListBox edit style.
attribute	An attribute for the DropDownListBox column. Attributes and their settings are listed in the table below.
value	The value to be assigned to the attribute when you use Modify. For ddlb attributes, *value* cannot be a DataWindow painter expression.

Attribute	Value
AllowEdit	Whether the user can type a value as well as choose from the DropDownListBox's list. Values are: ♦ Yes — Typing is allowed ♦ No — (Default) Typing is not allowed
AutoHScroll	Whether the DropDownListBox automatically scrolls horizontally when the user enters or deletes data. Values are: ♦ Yes — (Default) Scroll horizontally automatically ♦ No — Do not scroll automatically
Case	The case of the text in the DropDownListBox. Values are: ♦ Any — Character of any case allowed ♦ Upper — Characters converted to uppercase ♦ Lower — Characters converted to lowercase
Limit	An integer from 0–32767 specifying the maximum number of characters that can be entered in the DropDownListBox. Zero means unlimited.
NilisNull	Whether to set the data value of the DropDownListBox to NULL when the user leaves the edit box blank. Values are: ♦ Yes — Make the Empty string NULL ♦ No — Do not make the empty string NULL
Required	Whether the column is required. Values are: ♦ Yes — Required ♦ No — (Default) Not required

Attribute	Value
ShowList	Whether the ListBox portion of the DropDownListBox displays when the column has focus. A down arrow does not display at the right end of the DropDownListBox when ddlb.ShowList is yes. Values are: ♦ Yes — Display the list whenever the column has the focus ♦ No — Do not display the list until the user selects the column
Sorted	Whether the list in the DropDownListBox is sorted. Values are: ♦ Yes — The list is sorted ♦ No — The list is not sorted
UseAsBorder	Whether a down arrow displays at the right end of the DropDownListBox. Values are: ♦ Yes — Display the arrow ♦ No — Do not display the arrow Note that if ShowList is set to Yes, the column ignores the UseAsBorder attribute and the arrow never displays.
VScrollBar	Whether a vertical scrollbar displays in the DropDownListBox for long lists. Values are: ♦ Yes — Display a vertical scrollbar ♦ No — Do not display a vertical scrollbar

Examples

```
ls_data = dw_1.Describe("emp_status.ddlb.AllowEdit")
dw_1.Modify("emp_status.ddlb.Case=Any")
dw_1.Modify("emp_status.ddlb.Limit=30")
```

Depth

Description: The depth of a 3D graph.

Applies to: Graph objects

Used in: Describe, Modify

Syntax "*graphname*.Depth { = '*depthpercent*' }"

Parameter	Description
graphname	The graph object within the DataWindow for which you want to set the depth.
depthpercent	(*exp*) An integer whose value is the depth of the graph, specified as a percentage of the graph's width. *Depthpercent* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.Depth")
dw_1.Modify("graph_1.Depth='70'")
```

Detail_Bottom_Margin

Description The size of the bottom margin of the DataWindow's detail area.

Applies to Style keyword

Used in SyntaxFromSQL

Syntax Style(Detail_Bottom_Margin = *value*)

Parameter	Description
value	An integer specifying the size of the bottom margin of the detail area in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
   'Style(...Detail_Bottom_Margin = 25 ...)',  &
   errstring)
```

Detail_Top_Margin

Description The size of the top margin of the DataWindow's detail area.

Applies to Style keyword

Used in SyntaxFromSQL

Syntax

Style(Detail_Top_Margin = *value*)

Parameter	Description
value	An integer specifying the size of the top margin of the detail area in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
   'Style(...Detail_Top_Margin = 25 ...)',  &
   errstring)
```

Detail.*attribute*

See *Bandname.attribute*

DispAttr.*fontattribute*

Description

Settings for the appearance of various text components of a graph.

Applies to

Attributes of Graph objects, as noted throughout this section

Used in

Describe, Modify

Syntax

"*graphname.attribute*.DispAttr.*fontattribute* { = *value* }"

Parameter	Description
graphname	The graph object in a DataWindow for which you want to get or set font appearance values.
attribute	A text component of the graph, such as an *Axis* keyword (Category, Series, or Values), Legend, Pie, or Title, specifying the graph component whose appearance you want to get or set. These attributes have their own entries in this Appendix. You can also set font attributes for the label of an axis with the following syntax: "*graphname.axis*.LabelDispAttr.*fontattribute* { = *value* }"
fontattribute	An attribute that controls the appearance of text in the graph. Attributes and their settings are listed in the table below.
value	The value to be assigned to *fontattribute* when you use Modify. *Value* can be a quoted DataWindow painter expression.

Attribute for DispAttr	Value
Alignment	(*exp*) The alignment of the text. Values are: ♦ 0 — Left ♦ 1 — Right ♦ 2 — Center
AutoSize	(*exp*) Whether the text element should be autosized according to the amount of text being displayed. Values are: ♦ 0 — Do not autosize ♦ 1 — Autosize
BackColor	(*exp*) A long value specifying the background color of the text.
DisplayExpression	An expression whose value is the label for the graph component. The default expression is the attribute containing the text for the graph component. The expression can include the text attribute and add other variable text.
Font.Escapement	(*exp*) An integer specifying the rotation for the baseline of the text in tenths of a degree. For example, a value of 450 rotates the text 45 degrees. 0 is horizontal.
Font.CharSet	(*exp*) An integer specifying the character set to be used. Values are: ♦ 0 — ANSI ♦ 1 — The default character set for the specified font ♦ 2 — Symbol ♦ 128 — Shift JIS ♦ 255 — OEM
Font.Face	(*exp*) A string specifying the name of the font face, such as Arial or Courier.
Font.Family	(*exp*) An integer specifying the font family. (Windows uses both face and family to determine which font to use.) Values are: ♦ 0 — AnyFont ♦ 1 — Roman ♦ 2 — Swiss ♦ 3 — Modern ♦ 4 — Script ♦ 5 — Decorative

Attribute for DispAttr	Value
Font.Height	(*exp*) An integer specifying the height of the text in the unit measure for the DataWindow. To specify size in points, specify a negative number.
Font.Italic	(*exp*) Whether the text should be italic. Values are: ♦ 0 — Not italic (default) ♦ 1 — Italic
Font.Orientation	Same as Escapement.
Font.Pitch	(*exp*) The pitch of the font. Values are: ♦ 0 — The default pitch for your system ♦ 1 — Fixed ♦ 2 — Variable
Font.Strikethrough	(*exp*) Whether the text should be crossed out. Values are: ♦ 0 — Not crossed out (default) ♦ 1 — Crossed out
Font.Underline	(*exp*) Whether the text should be underlined. Values are: ♦ 0 — Not underlined (default) ♦ 1 — Underlined
Font.Weight	(*exp*) An integer specifying the weight of the text; for example, 400 for normal or 700 for bold.
Font.Width	(*exp*) An integer specifying the width of the font in the unit of measure specified for the DataWindow. Width is usually unspecified, which results in a default width based on the other attributes.
Format	(*exp*) A string containing the display format for the text.
TextColor	(*exp*) A long specifying the color to be used for the text.

Examples

```
setting =  &
   dw_1.Describe("Category.LabelDispAttr.Font.Face")

dw_1.Modify("Category.LabelDispAttr.Font.Face='Arial'")

dw_1.Modify("Title.DispAttr.DisplayExpression="  &
   + "'title + ~"~n~" + Today()'")
```

Edit.*attribute*

Description

Settings that affect the appearance and behavior of columns whose edit style is Edit.

Applies to

Column objects

Used in

Describe and Modify (1st syntax), SyntaxFromSQL where noted (2nd syntax)

Syntax

Describe and Modify:

"*columnname*.Edit.*attribute* { = *value* }"

SyntaxFromSQL:

Column(Edit.*attribute*=*value*)

Parameter	Description
columnname	The column with the Edit edit style for which you want to get or set attribute values. You can specify the column name or a pound sign (#) and the column number.
attribute	An attribute for the column's Edit style. Attributes and their settings are listed in the table below. The table identifies the attributes you can use with SyntaxFromSQL.
value	The value to be assigned to the attribute when you use Modify or SyntaxFromSQL. For most Edit attributes, you cannot specify a DataWindow painter expression. The exception is Edit.Format.

Attribute for Edit	Value
AutoHScroll	Whether the edit box scrolls horizontally automatically when data is entered or deleted. Values are: ♦ Yes — Scroll horizontally automatically ♦ No — Do not scroll horizontally automatically You can use AutoHScroll with SyntaxFromSQL. The setting applies to all the columns in the generated syntax.

Attribute for Edit	Value
AutoSelect	Whether to select the contents of the edit box a automatically when it receives focus. Values are: ♦ Yes — Select automatically ♦ No — Do not select automatically You can use AutoSelect with SyntaxFromSQL. The setting applies to all the columns in the generated syntax.
AutoVScroll	Whether the edit box scrolls vertically automatically when data is entered or deleted. Values are: ♦ Yes — Scroll vertically automatically ♦ No — Do not scroll vertically automatically You can use AutoVScroll with SyntaxFromSQL. The setting applies to all the columns in the generated syntax.
Case	The case of the text in the edit control. Values are: ♦ Any — Character of any case allowed ♦ Upper — Characters converted to uppercase ♦ Lower — Characters converted to lowercase
CodeTable	Whether the column has a code table. Values are: ♦ Yes — Code table defined ♦ No — No code table defined
DisplayOnly	Whether the column is display only. Values are: ♦ Yes — Do not allow the user to enter data; make the column display only ♦ No — (Default) Allow the user to enter data
FocusRectangle	Whether a dotted rectangle (the focus rectangle) will surround the current row of the column when the column has focus. Values are: ♦ Yes — (Default) Display the focus rectangle ♦ No — Do not display the focus rectangle You can use FocusRectangle with SyntaxFromSQL. The setting applies to all the columns in the generated syntax.
Format	(*exp*) A string containing the display format of the edit box. The value for Format is quoted and can be a DataWindow painter expression.

Attribute for Edit	Value
HScrollBar	Whether a horizontal scrollbar displays in the edit box. Values are: ♦ Yes — Display the horizontal scrollbar ♦ No — Do not display the horizontal scrollbar
Limit	A number specifying the maximum number of characters (0 to 32,767) that the user can enter. 0 means unlimited.
Name	A string whose value is the name of the predefined edit style associated with the column. Named styles are defined in the Database painter and can be reused. Specifying a name that has not been previously defined associates the name with the column but does not define a new edit style.
NilIsNull	Whether to set the value of the edit control to NULL when the user leaves it blank. Values are: ♦ Yes — Make the Empty string NULL ♦ No — Do not make the empty string NULL
Password	Whether to assign secure display mode to the column. When the user enters characters, they display as asterisks (*). Values are: ♦ Yes — Assign secure-display mode to the column ♦ No — Do not assign secure-display mode to the column
Required	Whether the column is required. Values are: ♦ Yes — It is required ♦ No — It is not required
Style	(*Describe only*) Returns the edit style of the column.
ValidateCode	Whether the code table will be used to validate user-entered values. Values are: ♦ Yes — Use the code table ♦ No — Do not use the code table
VScrollBar	Whether a vertical scrollbar displays in the line edit. Values are: ♦ Yes — Display vertical scrollbars ♦ No — Do not display vertical scrollbars

Examples

```
setting = dw_1.Describe("emp_name.Edit.AutoHScroll")

dw_1.Modify("emp_name.Edit.Required=no")
```

EditMask.*attribute*

Description

Settings that affect the appearance and behavior of columns with the EditMask edit style.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.EditMask.*attribute* { = *value* }"

Parameter	Description
columnname	The column with the EditMask edit style for which you want to get or set attribute values. You can specify the column name or a pound sign (#) and the column number.
attribute	An attribute for the column's EditMask style. Attributes and their settings are listed in the table below.
value	The value to be assigned to the attribute when you use Modify. For EditMask attributes, you cannot specify a DataWindow painter expression.

Attribute for EditMask	Value
AutoSkip	Whether the EditMask will automatically skip to the next field when the maximum number of characters have been entered: ♦ Yes — Skip automatically ♦ No — Do not skip automatically
CodeTable	Whether the column has a code table. Values are: ♦ Yes — Code table defined ♦ No — No code table defined
FocusRectangle	Whether a dotted rectangle (the focus rectangle) will surround the current row of the column when the column has focus. Values are: ♦ Yes — (Default) Display the focus rectangle ♦ No — Do not display the focus rectangle
Mask	A string containing the edit mask for the column.

Attribute for EditMask	Value
ReadOnly	Whether the column is read only. This attribute is valid only if EditMask.Spin is set to yes. Values are: ♦ Yes — Do not allow the user to enter data; make the column read only ♦ No — (Default) Allow the user to enter data
Required	Whether the column is required. Values are: ♦ Yes — It is required ♦ No — It is not required
Spin	Whether the user can scroll through a list of possible values for the column with a spin control. Values are: ♦ Yes — Display a spin control ♦ No — (Default) Do not display a spin control
SpinIncr	An integer indicating the amount to increment the spin control's values. The default for numeric values is 1, for dates 1 year, and for time 1 minute. For columns that are not numeric, date, or time, the spin control scrolls through values in an associated code table. If the EditMask.CodeTable attribute is No, the spin increment has no effect for these columns.
SpinRange	A string containing the maximum and minimum values for the column that will display in the spin control. The two values are separated by a tilde (~).This attribute is effective only if EditMaskSpin is Yes. Because the SpinRange string is within another quoted string, the tilde separator becomes four tildes, which reduces to a single tilde when parsed. The format for the string is: `"EditMask.SpinRange='minval~~~~maxval'"`

Examples

```
setting = dw_1.Describe("emp_status.EditMask.Spin")
dw_1.Modify("emp_bonus.EditMask.SpinIncr=1000")
dw_1.Modify("emp_bonus.EditMask.SpinRange='0~~~~5000'")
```

Elevation

Description: The elevation in a 3D graph.

Applies to: Graph objects

Used in: Describe, Modify

Syntax: "*graphname*.Elevation { = '*integer*' }"

Parameter	Description
graphname	The name of the graph object in the DataWindow for which you want to get or set the elevation.
integer	(*exp*) An integer specifying the elevation of the graph. Elevation can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("graph_1.Elevation")
dw_1.Modify("graph_1.Elevation=35")
dw_1.Modify("graph_1.Elevation='10~tIf(...,20,30)'")
```

EllipseHeight

Description: The radius of the vertical part of the corners of a RoundRectangle.

Applies to: RoundRectangle objects

Used in: Describe, Modify

Syntax: "*rrectname*.EllipseHeight { = '*integer*' }"

Parameter	Description
graphname	The name of the RoundRectangle object in the DataWindow for which you want to get or set the ellipse height.
integer	(*exp*) An integer specifying the radius of the vertical part of the corners of a RoundRectangle in the DataWindow's unit of measure. EllipseHeight can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("rrect_1.EllipseHeight")
dw_1.Modify("rrect_1.EllipseHeight=35")
dw_1.Modify("rrect_1.EllipseHeight='10~tIf(...,20,30)'")
```

EllipseWidth

Description: The radius of the horizontal part of the corners of a RoundRectangle.

Applies to: RoundRectangle objects

Used in: Describe, Modify

Syntax: "*rrectname*.EllipseWidth { = '*integer*' }"

Parameter	Description
graphname	The name of the RoundRectangle object in the DataWindow for which you want to get or set the ellipse width.
integer	(*exp*) An integer specifying the radius of the horizontal part of the corners of a RoundRectangle in the DataWindow's unit of measure. EllipseHeight can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("rrect_1.EllipseWidth")
dw_1.Modify("rrect_1.EllipseWidth=35")
dw_1.Modify("rrect_1.EllipseWidth='10~tIf(...,20,30)'")
```

Filename

Description: The filename containing the bitmap for a bitmap object in the DataWindow.

Applies to: Bitmap objects

Used in: Describe, Modify

Syntax: "*bitmapname*.Filename { = '*filestring*' }"

Parameter	Description
bitmapname	The name of the bitmap object in the DataWindow for which you want to get or set the filename.
filestring	(*exp*) A string containing the name of the file that contains the bitmap. *Filestring* can be an quoted DataWindow painter expression. If you include the name of the file containing the bitmap in the executable for the application, PowerBuilder will always use that bitmap; you cannot use Modify to change the bitmap.

Examples

```
setting = dw_1.Describe("bitmap_1.Filename")
dw_1.Modify("bitmap_1.Filename='exclaim.bmp'")
```

Filter

Description

The filter expression for the DataWindow. Filters allow you to use DataWindow functions to limit or change the data that displays in the DataWindow object when the application executes.

Used in

Create (See Appendix B)

> **Setting a DataWindow's filter**
> To set the filter for an existing DataWindow, use the SetFilter function. To filter the DataWindow buffer based on the current filter, use the Filter function.

FirstRowOnPage

Description

The first row currently visible in the DataWindow.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.FirstRowOnPage"

Example

```
setting = dw_1.Describe("DataWindow.FirstRowOnPage")
```

Font.Bias

Description

The way fonts are manipulated in the DataWindow during execution.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Font.Bias { = *biasvalue* }"

Parameter	Description
biasvalue	An integer indicating how the fonts will be manipulated at execution. *Biasvalue* cannot be a DataWindow painter expression. Values are: ♦ 0 — As display fonts ♦ 1 — As printer fonts ♦ 2 — Neutral; no manipulation will take place

Examples

```
setting = dw_1.Describe("DataWindow.Font.Bias")
dw_1.Modify("DataWindow.Font.Bias=1")
```

Font.*attribute*

Description

Settings that control the appearance of fonts within a DataWindow, except for graphs, which have their own settings (see DispAttr).

Applies to

Column, Computed Field, or Text objects

Used in

Describe and Modify (1st syntax), SyntaxFromSQL (2nd and 3rd syntax)

Syntax

Describe and Modify:

"*objectname*.Font.*attribute* { = '*value*' }"

SyntaxFromSQL:

Column(Font.*attribute=value*)

Text(Font.*attribute=value*)

Parameter	Description
objectname	The name of a column, computed field, or text object for which you want to get or set font attributes. For a column, you can specify its name or a pound sign (#) followed by the column number. When generating DataWindow syntax with SyntaxFromSQL, the Font settings apply to all columns or all text objects.
attribute	An attribute of the text. The attributes and their values are listed in the table below.

Parameter	Description
value	The value to be assigned to the attribute when you use Modify. *Value* can be a quoted DataWindow painter expression.

Attribute for Font	Value
CharSet	(*exp*) An integer specifying the character set to be used. Values are: ♦ 0 — ANSI ♦ 1 — The default character set for the specified font ♦ 2 — Symbol ♦ 128 — Shift JIS ♦ 255 — OEM
Escapement	(*exp*) An integer specifying the rotation for the baseline of the text in tenths of a degree. For example, a value of 450 rotates the text 45 degrees. 0 is horizontal.
Face	(*exp*) A string specifying the name of the font face, such as Arial or Courier.
Family	(*exp*) An integer specifying the font family. (Windows uses both face and family to determine which font to use.) Values are: ♦ 0 — AnyFont ♦ 1 — Roman ♦ 2 — Swiss ♦ 3 — Modern ♦ 4 — Script ♦ 5 — Decorative
Height	(*exp*) An integer specifying the height of the text in the unit measure for the DataWindow. To specify size in points, specify a negative number.
Italic	(*exp*) Whether the text should be italic. The default is no.
Pitch	(*exp*) The pitch of the font. Values are: ♦ 0 — The default pitch for your system ♦ 1 — Fixed ♦ 2 — Variable

Attribute for Font	Value
Strikethrough	(*exp*) Whether the text should be crossed out. The default is no.
Underline	(*exp*) Whether the text should be underlined. The default is no.
Weight	(*exp*) An integer specifying the weight of the text; for example, 400 for normal or 700 for bold.
Width	(*exp*) An integer specifying the width of the font in the unit of measure specified for the DataWindow. Width is usually unspecified, which results in a default width based on the other attributes.

Examples

```
dw_1.Describe("emp_name_t.Font.Face")
dw_1.Modify("emp_name_t.Font.Face='Arial'")
```

Footer.*attribute*

See *Bandname.attribute*

Format

Description

The display format for a column.

You can use the GetFormat and SetFormat functions instead of Describe and Modify to get and change a column's display format. The advantage to using Modify is the ability to specify an expression.

Applies to

Column, Computed Field objects

Used in

Describe, Modify

Syntax

"*objectname*.Format { = '*value* ' }"

Parameter	Description
objectname	The name of the column or computed field for which you want to get or set the display format.
value	(*exp*) A string specifying the display format. See the *User's Guide* for information on constructing display formats. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("phone.Format")

dw_1.Modify(  &
    "phone.Format='[red](@@@)@@@-@@@@;~~~'None~~~''")
```

GraphType

Description

The type of graph, for example, bar, pie, column, and so on.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.GraphType { = '*typeinteger*' }"

Parameter	Description
graphname	The graph object for which you want to get or change the type.
typeinteger	(*exp*) An integer identifying the type of graph in the DataWindow object. *Typeinteger* can be a quoted DataWindow painter expression. Values are: ♦ 1 — Area ♦ 2 — Bar ♦ 3 — Bar3D ♦ 4 — Bar3DObj ♦ 5 — BarStacked ♦ 6 — BarStacked3DObj ♦ 7 — Col ♦ 8 — Col3D ♦ 9 — Col3DObj ♦ 10 — ColStacked ♦ 11 — ColStacked3DObj ♦ 12 — Line ♦ 13 — Pie ♦ 14 — Scatter ♦ 15 — Area3D ♦ 16 — Line3D ♦ 17 — Pie3D

Examples

```
setting = dw_1.Describe("graph_1.GraphType")
dw_1.Modify("graph_1.GraphType=17")
```

Grid.ColumnMove

Description: Whether the user can rearrange columns by dragging.

Applies to: DataWindow

Used in: Describe, Modify

Syntax: "DataWindow.Grid.ColumnMove { = value } "

Parameter	Description
value	Whether the user can rearrange columns. Values are: ♦ Yes — The user can drag columns ♦ No — The user cannot drag columns

Examples

```
setting = dw_1.Describe("DataWindow.Grid.ColumnMove")
dw_1.Modify("DataWindow.Grid.ColumnMove=No")
```

Grid.Lines

Description: The way grid lines display and print in a DataWindow whose presentation style is grid.

Applies to: DataWindow

Used in: Describe, Modify

Syntax: "DataWindow.Grid.Lines { = *value* }"

Parameter	Description
value	An integer specifying whether grid lines are displayed on the screen and printed when you use Modify. Values are: ♦ 0 — Yes, grid lines are displayed and printed ♦ 1 — No, grid lines are not displayed and printed ♦ 2 — Grid lines are displayed, but not printed ♦ 3 — Grid lines are printed, but not displayed

Examples

```
setting = dw_1.Describe("DataWindow.Grid.Lines")
dw_1.Modify("DataWindow.Grid.Lines=2")
```

Header_Bottom_Margin

Description

The size of the bottom margin of the DataWindow's header area. Header_Bottom_Margin is meaningful *only* when Type is Grid or Tabular.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Header_Bottom_Margin = *value*)

Parameter	Description
value	An integer specifying the size of the bottom margin of the header area in the units specified for the DataWindow. The bottom margin is the distance between the bottom of the header area and the last line of the header.

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
   'Style(...Header_Bottom_Margin = 25 ...)',  &
   errstring)
```

Header_Top_Margin

Description

The size of the top margin of the DataWindow's header area. Header_Top_Margin is meaningful *only* when Type is Grid or Tabular.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Header_Top_Margin = *value*)

Parameter	Description
value	An integer specifying the size of the top margin of the header area in the units specified for the DataWindow. The top margin is the distance between the top of the header area and the first line of the header.

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
   'Style(...Header_Top_Margin = 500 ...)', errstring)
```

Header.*attribute*

See *Bandname.attribute*

Header.# .*attribute*

See *Bandname.attribute*

Height

Description — The height of an object in the DataWindow.

Applies to — Bitmap, Column, Computed Field, Graph, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in — Describe, Modify

Syntax — "*objectname*.Height { = '*value* ' }"

Parameter	Description
objectname	The object within the DataWindow whose height you want to get or set.
value	(*exp*) An integer specifying the height of the object in the unit of measure specified for the DataWindow. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("empname.Height")

dw_1.Modify("empname.Height=50")
```

Height.AutoSize

Description — Whether the object's width should be held constant and its height adjusted so that all the data is visible.

> **Minimum height**
> The height of the column, computed field, or text will never be less than the minimum height (the height selected in the painter).

Applies to — Column, Computed Field, Text objects

Used in — Describe, Modify

Syntax

"*objectname*.Height.AutoSize { = *value* }"

Parameter	Description
objectname	The object for which you want to get or set the AutoSize attribute.
value	Whether the width or height of the object will be adjusted to display all the data. The height is limited to what can fit on the page. Values are: ♦ No — Use the height defined in the painter and change the width so that all data is visible ♦ Yes — Use the width defined in the painter and calculate the height so that all the data is visible

Examples

```
setting = dw_1.Describe("empname.Height.AutoSize")
dw_1.Modify("empname.Height.AutoSize=Yes")
```

Help.*attribute*

Description

Settings for customizing the Help topics associated with DataWindow dialogs. For more information about Help, see the ShowHelp function in Chapter 1, "PowerScript Functions."

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Help.*attribute* {= *value* }"

Parameter	Description
attribute	An attribute for specifying DataWindow help. Help attributes and their settings are listed in the table below. The File attribute must have a valid filename before the rest of the Help attribute settings are valid.
value	The value to be assigned to the attribute when you use Modify. For Help attributes, *value* cannot be a DataWindow painter expression.

Attribute for Help	Value
Command	An integer specifying the type of help command that is specified in the following TypeID attributes. Values are: ♦ 0 — Index ♦ 1 — TopicID ♦ 2 — Search keyword
File	A string containing the fully qualified name of the compiled Help file (for example, C:\PB30\MYHELP.HLP). When this attribute has a value, Help buttons display on the DataWindow dialogs during execution.
TypeID	A string specifying the default help command to be used when a Help topic is not specified for the dialog using one of the following dialog-specific attributes.
TypeID.-ImportFile	A string specifying the Help topic for the Import File dialog, which may display when the ImportFile function is called in a script.
TypeID.Retrieve.-Argument	A string specifying the Help topic for the Retrieval Arguments dialog, which displays when retrieval arguments expected by the DataWindow's SELECT statement are not specified for the Retrieve function in a script.
TypeID.Retrieve.-Criteria	A string specifying the Help topic for the Prompt for Criteria dialog, which displays when the Criteria attributes have been turned on for at least one column and the Retrieve function is called in a script.
TypeID.SaveAs	A string specifying the Help topic for the Save As dialog, which may display when the Save As function is called in a script.
TypeID.-SetCrosstab	A string specifying the Help topic for the Crosstab Definition dialog, which may display when the CrosstabDialog function is called in a script.
TypeID.SetFilter	A string specifying the Help topic for the Set Filter dialog, which may display when the SetFilter and Filter functions are called in a script.
TypeID.SetSort	A string specifying the Help topic for the Set Sort dialog, which may display when the SetSort and Sort functions are called in a script.
TypeID.-SetSortExpr	A string specifying the Help topic for the Modify Expression dialog, which displays when the user double-clicks on a column in the Set Sort dialog.

Examples

```
setting = dw_1.Describe("DataWindow.Help.Command")

dw_1.Modify("DataWindow.Help.File='c"\pb40\myhelp.hlp'")

dw_1.Modify("DataWindow.Help.Command=1")

dw_1.Modify(  &
   "DataWindow.Help.TypeID.SetFilter='filter_topic'")

dw_1.Modify("DataWindow.Help.TypeID.Retrieve.Criteria" &
   + " = 'criteria_topic'")
```

Horizontal_Spread

Description

The space between columns in the detail area of the DataWindow object. Horizontal_Spread is meaningful *only* when Type is Grid or Tabular.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Horizontal_Spread = *value*)

Parameter	Description
value	An integer specifying the space between columns in the detail area of the DataWindow object area in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring, &
   'Style(...Horizontal_Spread = 25 ...)', errstring)
```

HorizontalScrollMaximum

Description

The maximum width of the scroll box of the DataWindow's horizontal scrollbar. This value is set by PowerBuilder based on the layout of the DataWindow object and the size of the DataWindow control. Use HorizontalScrollMaximum with HorizontalScrollPosition to synchronize horizontal scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.HorizontalScrollMaximum"

Example

```
setting =  &
   dw_1.Describe("DataWindow.HorizontalScrollMaximum")
```

HorizontalScrollMaximum2

Description

The maximum width of the second scroll box when the horizontal scrollbar is split (HorizontalScrollSplit is greater than 0). This value is set by PowerBuilder based on the content of the DataWindow. Use HorizontalScrollMaximum2 with HorizontalScrollPosition2 to synchronize horizontal scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.HorizontalScrollMaximum2"

Example

```
setting =  &
   dw_1.Describe("DataWindow.HorizontalScrollMaximum2")
```

HorizontalScrollPosition

Description

The position of the scroll box in the horizontal scrollbar. Use HorizontalScrollMaximum with HorizontalScrollPosition to synchronize horizontal scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.HorizontalScrollPosition { = *scrollvalue* }"

Parameter	Description
scrollvalue	An integer specifying the position of the scroll box in the horizontal scrollbar of the DataWindow

Example

```
smax1 = dw_1.Describe(  &
   "DataWindow.HorizontalScrollMaximum")
spos1 = dw_1.Describe(  &
   "DataWindow.HorizontalScrollPosition")
smax2 = dw_2.Describe(  &
   "DataWindow.HorizontalScrollMaximum")
pos2 = Integer(spos1) * Integer(smax2) / Integer(smax1)
modstring = "DataWindow.HorizontalScrollPosition="  &
   + String(pos2)
dw_1.Modify(modstring)
```

HorizontalScrollPosition2

Description

The position of the scroll box in the second portion of the horizontal scrollbar when the scrollbar is split (HorizontalScrollSplit is greater than 0). Use HorizontalScrollMaximum2 with HorizontalScrollPosition2 to synchronize horizontal scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.HorizontalScrollPosition2 { = *scrollvalue* }"

Parameter	Description
scrollvalue	An integer specifying the position of the scroll box in the second portion of a split horizontal scrollbar of the DataWindow

Examples

```
spos = dw_1.Describe(  &
    "DataWindow.HorizontalScrollPosition2")

dw_1.Modify("DataWindow.HorizontalScrollPosition2=200")
```

HorizontalScrollSplit

Description

The position of the split in the DataWindow's horizontal scrollbar. If HorizontalScrollSplit is zero, the scrollbar is not split.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.HorizontalScrollSplit { = *splitdistance* }"

Parameter	Description
splitdistance	An integer indicating where the split will occur in the horizontal scrollbar in a DataWindow object in the unit of measure specified for the DataWindow object

Examples

```
setting =  &
    dw_1.Describe("DataWindow.HorizontalScrollSplit")

dw_1.Modify("DataWindow.HorizontalScrollSplit=250")
```

ID

Description: The number of the column or TableBlob.

Applies to: Column or TableBlob objects

Used in: Describe

Syntax: "*objectname*.ID"

Parameter	Description
objectname	The name of the column or TableBlob for which you want the ID number

Example:

```
setting = dw_1.Describe("empname.ID")
```

Initial

Description: The initial value of the column in a newly inserted row.

Applies to: Column objects

Used in: Describe, Modify

Syntax: "*columnname*.Initial { ='*initialvalue* ' }"

Parameter	Description
initialvalue	A string containing the initial value of the column. Special values include: ♦ Empty — A string of length 0 ♦ Null — No value ♦ Spaces — All blanks ♦ Today — Current date, time, or date and time

Examples:

```
setting = dw_1.Describe("empname.Initial")
dw_1.Modify(""empname.Initial='empty'")
dw_1.Modify(""empstatus.Initial='A'")
```

Invert

Description: The way the colors in a bitmap object are displayed, either inverted or normal.

Applies to

Bitmap objects

Used in

Describe, Modify

Syntax

"*bitmapname*.Invert { = '*number*' }"

Parameter	Description
number	(*exp*) A boolean number indicating whether the colors of the bitmap will display inverted. Values are: ♦ 0 — (Default) No; do not invert the bitmap's colors ♦ 1 — Yes; display the bitmap with colors inverted *Number* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("bitmap_1.Invert")

dw_1.Modify(  &
    "bitmap_1.Invert='0~tIf(empstatus=~~~'A~~~',0,1)'")
```

Key

Description

Whether the column is part of the database table's primary key.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.Key { = *value* }"

Parameter	Description
columnname	The column for which you want to get or set primary key status.
value	Whether the column is part of the primary key. Values are: ♦ Yes — The column is part of the primary key ♦ No — The column is not part of the key

Examples

```
setting = dw_1.Describe("empid.Key")

dw_1.Modify("empid.Key=Yes")
```

KeyClause

Description: An expression to be used as the key clause when retrieving the blob.

Applies to: TableBlob objects

Used in: Describe, Modify

Syntax: "*tblobname*.KeyClause='*keyclause*' "

Parameter	Description
tblobname	The name of the TableBlob for which you want to specify a key clause.
keyclause	(*exp*) A string that will be built into a key clause using the substitutions provided. The key clause can be any valid WHERE clause. *Keyclause* can be a quoted DataWindow painter expression.

Example: With the following setting, the value of key_col will be put in col2 when PowerBuilder constructs the WHERE clause for the SELECTBLOB statement.

```
dw_1.Modify(blob_1.KeyClause='Key_col = :col2'")
```

Label.*attribute*

Description: Settings for a DataWindow whose presentation style is Label.

Applies to: DataWindow

Used in: Describe and Modify, except where noted (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax: Describe and Modify:

"DataWindow.Label.*attribute* { = *value* }"

SyntaxFromSQL:

DataWindow(Label.*attribute* =*value*)

Parameter	Description
attribute	An attribute for the Label presentation style. Attributes and their settings are listed in the table below.

Parameter	Description
value	The value to be assigned to the attribute when you use Modify. For Label attributes, *value* cannot be a DataWindow painter expression.

Attribute for Label	Value
Columns	An integer indicating the number of columns of labels on a sheet.
Columns.Spacing	An integer indicating the space between columns of labels in the units specified for the DataWindow object.
Ellipse_Height	An integer specifying the radius of the vertical part of the label in the unit of measure specified for the DataWindow object.
Ellipse_Width	An integer radius of the horizontal part of the label in the unit of measure specified for the DataWindow object.
Height	An integer specifying the height of a label in the units specified for the DataWindow object.
Name	A string containing the name of a label.
Rows	An integer indicating the number rows of labels on a sheet.
Rows.Spacing	An integer indicating the space between rows of labels on a sheet in the units specified for the DataWindow object.
Shape	A string specifying the shape of a label. Values are: ♦ Rectangle ♦ RoundRectangle ♦ Oval
Sheet	(*Describe only*) Whether the paper is sheet fed or continuous. Values are: ♦ Yes — Sheet fed ♦ No — Continuous
TopDown	(*Describe only*) Whether the labels will be printed from the top to the bottom or across the page. Values are: ♦ No — Print labels across the page ♦ Yes — Print labels from top to bottom
Width	An integer specifying the width of a label in the units specified for the DataWindow object.

Examples

```
setting = dw_1.Describe("DataWindow.Label.Sheet")
dw_1.Modify("DataWindow.Label.Width=250")
dw_1.Modify("DataWindow.Label.Height=150")
dw_1.Modify("DataWindow.Label.Columns=2")
dw_1.Modify("DataWindow.Label.Width=250")
dw_1.Modify("DataWindow.Label.Name='Address1'")
```

LabelDispAttr.*fontattribute*

See DispAttr.*fontattribute*

LastRowOnPage

Description: The last row currently visible in the DataWindow.

Applies to: DataWindow

Used in: Describe

Syntax: "DataWindow.LastRowOnPage"

Examples

```
setting = dw_1.Describe("DataWindow.LastRowOnPage")
```

Left_Margin

Description: The size of the left margin of the DataWindow object.

Applies to: Style keyword

Used in: SyntaxFromSQL

Syntax: Style(Left_Margin = *value*)

Parameter	Description
value	An integer specifying the size of the left margin in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring, &
    'Style(...LeftMargin = 500 ...)', errstring)
```

Legend

Description: The location of the legend in a graph object in a DataWindow.

Applies to: Graph objects

Used in: Describe, Modify

Syntax: "*graphname*.Legend { = '*value*' }"

Parameter	Description
graphname	The name of the graph object for which you want to specify the location of the legend.
value	(*exp*) A number indicating the location of the legend of a graph. Values are: ♦ 0 — None ♦ 1 — Left ♦ 2 — Right ♦ 3 — Top ♦ 4 — Bottom *Value* can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("graph_1.Legend")
dw_1.Modify("graph_1.Legend=2")
dw_1.Modify("graph_1.Legend='2~tIf(dept_id=200,0,2)'")
```

Legend.DispAttr.*fontattribute*

See DispAttr.*fontattribute*

Level

Description: The grouping level.

Applies to: Group keyword

Used in: Create, as shown in DataWindow syntax (Appendix B)

Syntax: Group(BY(*colnum1*,*colnum2*,...)... Level=*n* ...)

Message.Title

Description

The title of the dialog box that displays when an error occurs.

Applies to

DataWindow

Used in

Describe and Modify (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax

Describe and Modify:

"DataWindow.Message.Title { = '*titlestring* ' }"

SyntaxFromSQL:

DataWindow(Message.Title='*titlestring* ')

Parameter	Description
titlestring	A string containing the title that displays in the title bar of the DataWindow dialog box that displays when an error occurs.

Examples

```
setting = dw_1.Describe("DataWindow.Message.Title")

dw_1.Modify("DataWindow.Message.Title='Bad, Bad, Bad'")

SQLCA.SyntaxFromSQL(sql_syntax,  &
   "Style(...) &
DataWindow(Message.Title='Sales Report' ...) ...",  &
   ls_Errors)
```

Moveable

Description

Whether the specified object in the DataWindow can be moved during execution. Movable objects should be in the DataWindow's foreground.

Applies to

Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, and Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Moveable { = *number* }"

Parameter	Description
objectname	The object within the DataWindow for which you want to get or set its Moveable attribute, that is, whether the user can move the object.

Parameter	Description
number	An boolean number specifying whether the object is moveable. Values are: ♦ 0 — False, the object is not moveable ♦ 1 — True, the object is moveable

Examples

```
setting = dw_1.Describe("bitmap_1.Moveable")
dw_1.Modify(""bitmap_1.Moveable=1")
```

Name

Description: The name of the column.

Applies to: Column objects

Used in: Describe

Syntax: "*columnname*.Name"

Parameter	Description
columnname	The column for which you want the name. You can specify the column number preceded by #.

Example

```
setting = dw_1.Describe("#4.Name")
```

Nest_Arguments

Description: The retrieval arguments for a nested report. The number of arguments in the list should match the number of retrieval arguments defined for the nested report.

Applies to: Report objects

Used in: Describe, Modify

Syntax: "*reportname*.Nest_Arguments { = *list* } "

Parameter	Description
reportname	The name of the nested report for which you want to set retrieval arguments.

Parameter	Description
list	A list of retrieval arguments for the nested report. The format for the list is: (("*arg1*") {,("*arg2*") {,("*arg3*") {,...}}}) The list is not a quoted string. It is surrounded by parentheses and each argument within the list is parenthesized, surrounded with double quotes, and separated by commas. If an argument is a literal string, use single quotes within the double quotes.

Example

```
setting = dw_1.Describe("rpt_1.Nest_Arguments")

dw_1.Modify("rpt_1.Nest_Arguments"  &
    "=((~"cust_id~"),(~"'Eastern'~"))")
```

Nested

Description — Whether the DataWindow contains nested DataWindows. Values returned are Yes or No.

Applies to — DataWindow

Used in — Describe

Syntax — "DataWindow.Nested"

Example

```
setting = dw_1.Describe("DataWindow.Nested")
```

NewPage (Group keyword)

Description — Whether a change in the value of a group column causes a page break.

Applies to — Group keyword

Used in — SyntaxFromSQL

Syntax — Group(*colnum1*, *colnum2* NewPage)

Example

```
SQLCA.SyntaxFromSQL(sql_syntax,  &
    "Style(Type=Group) " +  &
    "Group(dept_id NewPage ResetPageCount)",  &
    ls_Errors)
```

NewPage (Report objects)

Description

Whether a nested report starts on a new page. NewPage only applies to reports in a composite DataWindow. Note that if the Trail_Footer attribute of the preceding report is set to No, the current report will be forced to begin on a new page regardless of the NewPage value.

Applies to

Report objects

Used in

Describe, Modify

Syntax

"*reportname*.NewPage { = *value* } "

Parameter	Description
reportname	The name of the report object for which you want to get or set its NewPage attribute
value	Whether the report begins a new page. Values are: ♦ Yes — Start the report on a new page ♦ No — Do not start the report on a new page

Example

```
SQLCA.SyntaxFromSQL(sql_syntax,  &
   "Style(Type=Group) " +  &
   "Group(dept_id NewPage ResetPageCount)",  &
   ls_Errors)
```

Objects

Description

A list of the objects in the DataWindow object. If an object doesn't have a name, PowerBuilder assigns it an arbitrary name. The names are returned as a tab-separated list.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.Objects"

Example

```
setting = dw_1.Describe("DataWindow.Objects")
```

OLE.Client.*attribute*

Description

Settings that some OLE server applications use to identify the client's information. The attribute values may be used to construct the title of the server window.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.OLE.Client.*attribute* {='*value* ' }"

Parameter	Description
attribute	An OLE client attribute, as shown in the table below.
value	Values for the attributes are shown below. *Value* cannot be a DataWindow painter expression.

Attribute for OLE.Client	Value
Class	The client class for the DataWindow. The default is DataWindow.
Name	The client name for the DataWindow. The default is Untitled.

Examples

```
ls_data = dw_1.Describe("DataWindow.OLE.Client.Class")
dw_1.Modify("DataWindow.OLE.Client.Class = 'PB'")
```

OLEClass

Description

The name of the OLE class for the TableBlob object.

Applies to

TableBlob objects

Used in

Describe, Modify

Syntax

"*tblobname*.OLEClass { = '*oleclassname* ' }"

Parameter	Description
tblobname	The TableBlob column for which you want to get or set the class of server application.
oleclassname	(*exp*) A string specifying a class of an OLE server application installed on your system. *Oleclassname* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("blob_1.OLEClass")
dw_1.Modify("blob_1.OLEClass='Word.Document'")
```

OverlapPercent

Description

The percentage of overlap for the data markers (such as bars or columns) in different series in a graph.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.OverlapPercent { = '*integer*' }"

Parameter	Description
graphname	The name of the graph object in the DataWindow object for which you want to get or set the percentage of overlap.
integer	(*exp*) An integer specifying the percent of the width of the data markers that will overlap when you use Modify. *Integer* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.OverlapPercent")
dw_1.Modify("graph_1.OverlapPercent=25")
```

Pen.*attribute*

Description

Settings for a line or the outline of an object.

Applies to

Line, Oval, Rectangle, and RoundRectangle objects

Used in

Describe, Modify

Syntax

"*objectname*.Pen.*attribute* { = *value* }"

Parameter	Description
objectname	The name of the object whose Pen attribute you want to get or set.
attribute	An attribute that applies to the Pen characteristics of *objectname*, as listed in the Attribute table below.

Parameter	Description
value	The value of the attribute when you use Modify, as shown below. *Value* can be a quoted DataWindow painter expression.

Attribute for Pen	Value
Color	(*exp*) A long specifying the color (the red, green, and blue values) to be used as the object's line color.
Style	(*exp*) A number specifying the style of the line. Values are: ♦ 0 — Solid ♦ 1 — Dash ♦ 2 — Dotted ♦ 3 — Dash-dot pattern ♦ 4 — Dash-dot-dot pattern ♦ 5 — Null (no visible line)
Width	(*exp*) A number specifying the width of the line in the unit of measure specified for the DataWindow.

Examples

```
setting = dw_1.Describe("line_1.Pen.Width")
dw_1.Modify("line_1.Pen.Width=10")
```

Perspective

Description

The distance the graph appears from the front of the window.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.Perspective { = '*integer*' }"

Parameter	Description
graphname	The name of the graph object in the DataWindow object for which you want to get or set the perspective.

Parameter	Description
integer	(*exp*) An integer between 1 and 100 specifying how far away the graph appears when you use Modify. The larger the number, the greater the distance and the smaller the graph appears. *Integer* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.Perspective")
dw_1.Modify("graph_1.Perspective=20")
```

Pie.DispAttr.*fontattribute*

See DispAttr.*fontattribute*

Pointer

Description

The image to be used for the mouse pointer when the pointer is over the specified object. If you specify a pointer for the whole DataWindow, PowerBuilder uses that pointer except when the pointer is over an object that also has a Pointer setting.

Applies to

DataWindow, Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, and Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Pointer { = '*pointername* ' }"

Parameter	Description
objectname	The name of the object in the DataWindow for which you want to get or set the pointer. Specify DataWindow to specify the pointer for the whole DataWindow.
pointername	(*exp*) A string specifying a value of the Pointer enumerated data type or the name of a cursor file (.CUR) to be used for the pointer. (See the SetPointer function for a list of Pointer values.) *Pointername* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.Pointer")
dw_1.Modify("graph_1.Pointer = 'Cross!'")
dw_1.Modify("graph_1.Pointer = 'c:\pb040\mycurs.cur'")
```

Print.*attribute*

Description Attributes that control the printing of a DataWindow.

Applies to DataWindow

Used in Describe and Modify (1st syntax), SyntaxFromSQL where noted (2nd syntax)

Syntax Describe and Modify:

```
"DataWindow.Print.attribute { = value }"
```

SyntaxFromSQL:

```
DataWindow(Print.attribute=value)
```

Parameter	Description
attribute	An attribute for printing. Attributes and their settings are listed in the table below.
value	The value to be assigned to the attribute when you use Modify. *Value* cannot be a DataWindow painter expression.

Attribute for Print	Value
Collate	Whether printing is collated. Note that collating is usually slower since the print is repeated to produce collated sets. Values are: ♦ Yes — Collate the pages of the print job ♦ No — (Default) Do not collate
Color	An integer indicating whether the printed output will be color or monochrome. Values are: ♦ 1 — Color ♦ 2 — Monochrome
Columns	An integer specifying the number of newspaper-style columns the DataWindow will print on a page. For purposes of page fitting, the whole DataWindow is a single column. The default is 1.
Columns.Width	An integer specifying the width of the newspaper-style columns in the units specified for the DataWindow.
Copies	An integer indicating the number of copies to be printed.

Attribute for Print	Value
DocumentName	A string containing the name that will display in the print queue when the user sends the contents of the DataWindow object to the printer.
Duplex	An integer indicating the orientation of the printed output. Values are: ♦ 1 — Simplex (none) ♦ 2 — Horizontal ♦ 3 — Vertical
Filename	A string containing the name of the file to which you want to print the report. An empty string means send to the printer.
Margin.Bottom	An integer indicating the width of the bottom margin on the printed page in the units specified for the DataWindow. You can set Margin.Bottom when using SyntaxFromSQL to generate DataWindow syntax.
Margin.Left	An integer indicating the width of the left margin on the printed page in the units specified for the DataWindow. You can set Margin.Left when using SyntaxFromSQL to generate DataWindow syntax.
Margin.Right	An integer indicating the width of the right margin on the printed page in the units specified for the DataWindow. You can set Margin.Right when using SyntaxFromSQL to generate DataWindow syntax.
Margin.Top	An integer indicating the width of the top margin on the printed page in the units specified for the DataWindow. You can set Margin.Top when using SyntaxFromSQL to generate DataWindow syntax.
Orientation	An integer indicating the print orientation. Values are: ♦ 0 — The default orientation for your printer ♦ 1 — Landscape ♦ 2 — Portrait
Page.Range	A string containing the numbers of the pages you want to print, separated by commas. You can also specify a range with a dash. For example, to print pages 1, 2, and 5 through 10, enter: "1,2, 5-10". The empty string means print all.

Attribute for Print	Value
Page.-RangeInclude	An integer indicating what pages to print within the desired range. Values are: ♦ 0 — Print all ♦ 1 — Print all even pages ♦ 2 — Print all odd pages
Paper.Size	An integer indicating the size of the paper that will be used for the output: ♦ 0 — Default paper size for the printer ♦ 1 — Letter 8 1/2 x 11 in ♦ 2 — LetterSmall 8 1/2 x 11in ♦ 3 — Tabloid 17 x 11 inches ♦ 4 — Ledger 17 x 11 in ♦ 5 — Legal 8 1/2 x 14 in ♦ 6 — Statement 5 1/2 x 8 1/2 in ♦ 7 — Executive 7 1/4 x 10 1/2 in ♦ 8 — A3 297 x 420 mm ♦ 9 — A4 210 x 297 mm ♦ 10 — A4 Small 210 x 297 mm ♦ 11 — A5 148 x 210 mm ♦ 12 — B4 250 x 354 ♦ 13 — B5 182 x 257 mm ♦ 14 — Folio 8 1/2 x 13 in ♦ 15 — Quarto 215 x 275 mm ♦ 16 — 10x14 in ♦ 17 — 11x17 in ♦ 18 — Note 8 1/2 x 11 in ♦ 19 — Envelope #9 3 7/8 x 8 7/8 ♦ 20 — Envelope #10 4 1/8 x 9 1/2 ♦ 21 — Envelope #11 4 1/2 x 10 3/8 ♦ 22 — Envelope #12 4 x 11 1/276 ♦ 23 — Envelope #14 5 x 11 1/2 ♦ 24 — C size sheet ♦ 25 — D size sheet ♦ 26 — E size sheet ♦ 27 — Envelope DL 110 x 220mm

Attribute for Print	Value
	♦ 28 — Envelope C5 162 x 229 mm ♦ 29 — Envelope C3 324 x 458 mm ♦ 30 — Envelope C4 229 x 324 mm ♦ 31 — Envelope C6 114 x 162 mm ♦ 32 — Envelope C65 114 x 229 mm ♦ 33 — Envelope B4 250 x 353 mm ♦ 34 — Envelope B5 176 x 250 mm ♦ 35 — Envelope B6 176 x 125 mm ♦ 36 — Envelope 110 x 230 mm ♦ 37 — Envelope Monarch 3.875 x 7.5 in ♦ 38 — 6 3/4 Envelope 3 5/8 x 6 1/2 in ♦ 39 — US Std Fanfold 14 7/8 x 11 in ♦ 40 — German Std Fanfold 8 1/2 x 12 in ♦ 41 — German Legal Fanfold 8 1/2 x 13 in
Paper.Source	An integer indicating the bin that will be used as the paper source. Values are: ♦ 0 — Default ♦ 1 — Upper ♦ 2 — Lower ♦ 3 — Middle ♦ 4 — Manual ♦ 5 — Envelope ♦ 6 — Envelope manual ♦ 7 — Auto ♦ 8 — Tractor ♦ 9 — Smallfmt ♦ 10 — Largefmt ♦ 11 — Large capacity ♦ 14 — Cassette
Preview	Whether the DataWindow object is displayed in preview mode. Values are: ♦ Yes — Display in preview mode ♦ No — (Default) Do not display in preview mode

Attribute for Print	Value
Preview.Rulers	Whether the rulers display when the DataWindow object displays in preview mode: ♦ Yes — Display the rulers ♦ No — (Default) Do not display the rulers
Preview.Zoom	An integer indicating the zoom factor of the print preview. The default is 100%.
Prompt	Whether a prompt will display before the job prints so the use can cancel the print job. Values are: ♦ Yes — (Default) Display a prompt before the job prints ♦ No — Do not display a prompt before the job prints
Quality	An integer indicating the quality of the output. Values are: ♦ 0 — Default ♦ 1 — High ♦ 2 — Medium ♦ 3 — Low ♦ 4 — Draft
Scale	An integer specifying the scale of the printed output as a percent.

Examples

```
ls_data = dw_1.Describe("DataWindow.Print.Scale")
dw_1.Modify("DataWindow.Print.Paper.Size = 3")
dw_1.Modify("DataWindow.Print.Margin.Top=500")
```

Printer

Description — The name of the printer for printing the DataWindow as specified in the system's printer selection dialog.

Applies to — DataWindow

Used in — Describe

Syntax — "DataWindow.Printer"

Example

```
setting = dw_1.Describe("DataWindow.Printer")
```

Processing

Description

The type of processing required to display the data in the selected presentation style.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.Processing"

Return values are:

- 0 — (Default) Form, group, query, or tabular
- 1 — Grid
- 2 — Label
- 3 — Graph
- 4 — Crosstab

Example

```
setting = dw_1.Describe("DataWindow.Processing")
```

Protect

Description

The protection setting of a column. The Protect attribute overrides tab order settings. When a column is protected, the user cannot edit it even if the column's tab order is greater than 0.

Applies to

A column

Used in

Describe, Modify

Syntax

"*columnname*.Protect { = '*integer*' }"

Parameter	Description
columnname	The name of the column for which you want to get or set the protection.
integer	(*exp*) A boolean integer specifying whether the column is protected. Values are: ♦ 0 — False, the column is not protected ♦ 1 — True, the column is protected *Integer* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_stat.Protect")
dw_1.Modify("emp_stat.Protect=1")
dw_1.Modify("emp_stat.Protect='1-tIf(IsRowNew(),0,1)'")
```

QueryMode

Description

Whether the DataWindow is in query mode. In query mode, the user can specify the desired data by entering WHERE criteria in one or more columns. After the user specifies retrieval criteria in query mode, subsequent calls to Retrieve will use the new criteria.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.QueryMode { = *value* }"

Parameter	Description
value	Whether the DataWindow is in query mode. Values are: ♦ Yes — Query mode is enabled ♦ No — Query mode is disabled and the user's specification stored for the next call to Retrieve

Tip
Setting QuerySort to Yes also puts the DataWindow into Query mode.

Examples

```
setting = dw_1.Describe("DataWindow.QueryMode")
dw_1.Modify("DataWindow.QueryMode=yes")
```

QuerySort

Description

Whether the result set is sorted when the DataWindow retrieves the data specified in query mode. When query sort is on, the user specifies sorting criteria in the first row of the query form.

Applies to

DataWindow

Used in

Describe, Modify

Syntax "DataWindow.QueryMode { = *value* }"

Parameter	Description
value	Whether the data retrieved from query mode specifications is sorted. Values are: ♦ Yes — Sorting is enabled ♦ No — Sorting is disabled

Tip
If the DataWindow is not already in query mode, setting QuerySort to Yes also sets QueryMode to Yes, putting the DataWindow in query mode.

When you set QuerySort to No, the DataWindow remains in query mode until you also set QueryMode to No.

Examples

```
setting = dw_1.Describe("DataWindow.QuerySort")
dw_1.Modify("DataWindow.QuerySort=yes")
```

RadioButtons.*attribute*

Description Attributes that control the appearance and behavior of a column with the RadioButton edit style.

Applies to Column objects

Used in Describe, Modify

Syntax "*columnname*.RadioButtons.*attribute* { = *value* }"

Parameter	Description
columnname	The name of column that has the RadioButton edit style.
attribute	An attribute for the RadioButton column. Attributes and their settings are listed in the table below.
value	The value to be assigned to the attribute when you use Modify. For RadioButton attributes, *value* cannot be a DataWindow painter expression.

Attribute for RadioButtons	Value
3D	Whether the radio buttons are 3D. Values are: ♦ Yes — Make the buttons 3D ♦ No — Do not make the buttons 3D
Columns	An integer constant specifying the number of columns of radio buttons.
LeftText	Whether the text labels for the radio buttons are on the left side. Values are: ♦ Yes — The text is on the left of the radio buttons ♦ No — The text is on the right of the radio buttons
Scale	Whether the circle is scaled to the size of the font. Scale has an effect only when 3D is No. Values are: ♦ Yes — Scale the circles ♦ No — Do not scale the circles

Examples

```
setting = &
   dw_1.Describe("emp_gender.RadioButtons.LeftText")

dw_1.Modify("emp_gender.RadioButtons.LeftText=no")

dw_1.Modify("emp_gender.RadioButtons.3D=Yes")

dw_1.Modify("emp_gender.RadioButtons.Columns=2")
```

Range

Description

The rows in the DataWindow that are included in a graph.

Applies to

Graph objects

Used in

Describe

Syntax

"*graphname*.Range"

Parameter	Description
graphname	The name of the graph object within the DataWindow that will display the graphed rows

Possible values are:

♦ -1 — The rows on a single page in the DataWindow object

- 0 — All the rows in the DataWindow object
- n — The number of a group level in the DataWindow object

Example

```
setting = dw_1.Describe("DataWindow.Range")
```

ReadOnly

Description

Whether the DataWindow is read-only. (DataWindows defined in the Report painter are automatically read-only.)

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.ReadOnly { = value }"

Parameter	Description
value	Whether the DataWindow is read-only. Values are: ♦ Yes — Make the DataWindow read-only ♦ No — (Default) Do not make the DataWindow read-only

Examples

```
setting = dw_1.Describe("DataWindow.ReadOnly")
dw_1.Modify("DataWindow.ReadOnly=Yes")
```

Report

Description

Whether the DataWindow is a read-only report.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Report = *value*)

Parameter	Description
value	Whether the DataWindow is a read-only report, similar to a DataWindow created in the Report painter. Values are: ♦ Yes — The DataWindow is a read-only report ♦ No — The DataWindow is not read-only

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
   'Style(...Report = yes ...)', errstring)
```

ResetPageCount

Description: Specifies that a change in the value of the group column causes the page count to begin again at 0.

Applies to: Group keyword

Used in: SyntaxFromSQL

Syntax: Group(*col1* {*col2* ...} ... ResetPageCount)

Example:

```
SQLCA.SyntaxFromSQL(sql_syntax, &
    "Style(Type=Group) "   &
    + "Group(dept_id NewPage ResetPageCount)",   &
    errorvar)
```

Resizeable

Description: Whether the user can resize the specified object.

Applies to: Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, and Text objects

Used in: Describe, Modify

Syntax: "*objectname*.Resizeable { = *value* }"

Parameter	Description
objectname	The object within the DataWindow whose Resizeable setting you want to get or set.
value	A boolean number indicating whether *objectname* can be resized. Values are: ♦ 0 — (False) The object cannot be resized ♦ 1 — (True) The object can be resized

Examples:

```
setting = dw_1.Describe("graph_1.Resizeable")
dw_1.Modify("graph_1.Resizeable=1")
dw_1.Modify("bitmap_1.Resizeable=0")
```

Retrieve

Description: The SQL statement for the DataWindow.

Applies to: Table keyword

Used in Create, as shown in DataWindow syntax (Appendix B)

Syntax Table(... Retrieve=*selectstatement* ...)

Retrieve.AsNeeded

Description Whether rows will be retrieved only as needed from the database. After the application calls the Retrieve function to get enough rows to fill the visible portion of the DataWindow, additional rows are "needed" when the user scrolls down to view rows that haven't been viewed yet.

Applies to DataWindow

Used in Describe, Modify

Syntax "DataWindow.Retrieve.AsNeeded { = '*value* ' }"

Parameter	Description
value	Whether rows will be retrieved only as needed from the database. Values are: ♦ Yes — Rows will be retrieved only as needed ♦ No — All rows will be retrieved when the Retrieve function is called

Examples

```
setting = dw_1.Describe("DataWindow.Retrieve.AsNeeded")
dw_1.Modify("DataWindow.Retrieve.AsNeeded=Yes")
```

Rotation

Description The degree of left-to-right rotation for the graph object within the DataWindow when the graph has a 3D type.

Applies to Graph objects

Used in Describe, Modify

Syntax "*graphname*.Rotation = { '*integer* ' }"

Parameter	Description
graphname	The name of the graph object for which you want to get or set the rotation.

Parameter	Description
integer	(*exp*) The degree of rotation for the graph when you use Modify. Effective values range from -90 to 90. *Integer* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.Rotation")
dw_1.Modify("graph_1.Rotation=25")
dw_1.Modify("graph_1.Rotation='1~tHour(Now())'")
```

Row.Resize

Description

Whether the user can use the mouse to change the height of the rows in the detail area of the DataWindow.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Row.Resize { = value } "

Parameter	Description
value	Whether the user can resize the rows in the detail area. Values are: ♦ 1 — Yes, the user can resize the rows ♦ 0 — No, the user cannot resize the rows

Examples

```
setting = dw_1.Describe("DataWindow.Row.Resize")
dw_1.Modify("DataWindow.Row.Resize=0")
```

Rows_Per_Detail

Description

The number of rows in each column of an N-Up DataWindow object. This attribute is ignored unless the Type attribute for the Style keyword is tabular.

Applies to

DataWindow keyword

Used in

SyntaxFromSQL

Syntax

DataWindow(... Rows_Per_Detail=*n* ...)

Parameter	Description
n	A long specifying the number of rows in each column

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
    'DataWindow(...Rows_Per_Detail = 12 ...)',  &
    errstring)
```

Selected

Description

A list of selected objects within the DataWindow.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Selected = '*list*' "

Parameter	Description
list	A list of the objects you want to select when you use Modify. In the list, you designate a group of objects by specifying range of row numbers and a range of objects in the format: *startrow/endrow/startobject/endobject* To specify more that one group, separate each group with a semicolon. *startrow1/endrow1/startobj1/endobj1;startrow2/endrow2/startobj2/endobj2;...* Do not include spaces in the string. You must use column names not column numbers.

Examples

```
setting = dw_1.Describe("DataWindow.Selected")

dw_1.Modify("DataWindow.Selected="  &
    "'1/10/emp_id/emp_name;12/23/salary/status'")
```

Selected.Data

Description A list describing the selected data in the DataWindow. Each column's data is separated by a tab and each row is on a separate line.

Applies to DataWindow

Used in Describe

Syntax "DataWindow.Selected.Data"

Example

```
setting = dw_1.Describe("DataWindow.Selected.Data")
```

Selected.Mouse

Description Whether the user can use the mouse to select columns.

Applies to DataWindow

Used in Describe, Modify

Syntax "DataWindow.Selected.Mouse { = *value* }"

Parameter	Description
value	Whether the user can use the mouse to select columns. Values are: ♦ Yes — The mouse can be used ♦ No — The mouse cannot be used

Examples

```
setting = dw_1.Describe("DataWindow.Selected.Mouse")
dw_1.Modify("DataWindow.Selected.Mouse = Yes")
```

Series

See *Axis* and *Axis.attribute*

ShadeColor

Description The color used for shading the back edge of the series markers when the graph's type is 3D. ShadeColor has no effect unless Series.ShadeBackEdge is 1 (Yes).

Applies to Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.ShadeColor { = '*long* ' }"

Parameter	Description
long	(*exp*) A long specifying the color of the shading for 3D data markers. You can use the RGB function to calculate the desired long value. *Long* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.ShadeColor")

dw_1.Modify("graph_1.ShadeColor=16600000")

dw_1.Modify("graph_1.ShadeColor=" + &
   String(RGB(90,90,90)))

dw_1.Modify("graph_1.ShadeColor='0~t"  &
   + If(salary>50000,"  &
   + String(RGB(100,90,90))  &
   + ","  &
   + String(RGB(90,90,100))  &
   + ")'")
```

ShowDefinition

Description

Whether the DataWindow definition will display. The DataWindow will display the column names instead of data.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.ShowDefinition { = '*value* ' }"

Parameter	Description
value	(*exp*) Whether the column names will display. Values are: ♦ Yes — Display the column names ♦ No — Do not display the data, if any *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("DataWindow.ShowDefinition")

dw_1.Modify("DataWindow.ShowDefinition=Yes")
```

SizeToDisplay

Description: Whether the graph should be sized automatically to the display area.

Applies to: Graph objects

Used in: Describe, Modify

Syntax: "*graphname*.SizeToDisplay { = '*value*' }"

Parameter	Description
graphname	The graph object in the DataWindow for which you want to get or set its adjustability.
value	(*exp*) A boolean number specifying whether to adjust the size of the graph to the display. Values are: ♦ 0 — False, do not adjust the size of the graph ♦ 1 — True, adjust the size of the graph *Value* can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("graph_1.SizeToDisplay")
dw_1.Modify("graph_1.SizeToDisplay=0")
```

SlideLeft

Description: Whether the object moves to the left when other objects to the left leave empty space available.

Applies to: Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in: Describe, Modify

Syntax: "*objectname*.SlideLeft { = '*value* ' }"

Parameter	Description
objectname	The name of the object for which you want to get or set its Slide setting.
value	(*exp*) Whether the object slides left when there is empty space to its left. Values are: ♦ Yes — The object will slide left into available space ♦ No — The object will remain in position *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.SlideLeft")
dw_1.Modify("emp_lname.SlideLeft=yes")
```

SlideUp

Description

Whether the object moves up when other objects above it leave empty space available.

Applies to

Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.SlideUp { = '*value* ' }"

Parameter	Description
objectname	The name of the object for which you want to get or set its Slide setting.
value	(*exp*) How the object slides up when there is empty space to its left. Values are: ♦ AllAbove — Slide the object up if all the objects in the row above it are empty ♦ DirectlyAbove — Slide the column or object up if the objects directly above it are empty ♦ No — The object will not slide up *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.SlideUp")
dw_1.Modify("emp_lname.SlideUp=no")
```

Sort

Description

Sort criteria for a newly created DataWindow. To specify sorting for existing DataWindows, see the SetSort and Sort functions.

Applies to

Table keyword in DataWindow syntax

Used in

Create, as shown in Appendix B, "DataWindow Syntax Listing"

Syntax

Table(... Sort=*stringexpression* ...)

Parameter	Description
stringexpression	a string whose value is valid sort criteria. See the SetSort function for the format for sort criteria. If the criteria string is NULL, PowerBuilder prompts for a sort specification when it displays the DataWindow.

Spacing

Description

The gap between categories in a graph.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.Spacing { = '*integer*' }"

Parameter	Description
graphname	The name of the graph object in the DataWindow for which you want to get or set the spacing.
integer	(*exp*) An integer specifying the gap between categories in the graph. You specify the value as a percentage of the width of the data marker. For example, in a bar graph, 100 is the width of one bar; 50 is half a bar, and so on. *Integer* can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("graph_1.Spacing")
dw_1.Modify("graph_1.Spacing=120")
```

Sparse

Description

The names of repeating columns that will be suppressed in the DataWindow.

Applies to

DataWindow

Used in

Describe and Modify (Syntax 1), Create (Syntax 2)

Syntax

For Describe and Modify:

"DataWindow.Sparse { = '*list*' }"

Parameter	Description
list	(*exp*) A tab-separated list of column names to be suppressed. *List* can be a quoted DataWindow painter expression.

For Create, include the following at the end of the DataWindow syntax (see Appendix B, "DataWindow Syntax Listing")

Sparse(names="*col1*~t *col2*~t *col3* ...")

Examples

```
setting = dw_1.Describe("DataWindow.Sparse")
dw_1.Modify("DataWindow.Sparse='col1~tcol2'")
```

Storage

Description

The amount of virtual storage in bytes that has been allocated for the DataWindow object.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.Storage"

Canceling a query that uses too much storage
You can check this attribute in the script for the RetrieveRow event in the DataWindow control and cancel a query if it is consuming too much storage.

Example

```
setting = dw_1.Describe("DataWindow.Storage")
IF Long(setting) > 50000 THEN dw_1.SetActionCode(1)
```

Summary.*attribute*

See *Bandname.attribute*

Syntax

Description — The complete syntax for the DataWindow.

Applies to — DataWindow

Used in — Describe

Syntax — "DataWindow.Syntax"

Examples

```
setting = dw_1.Describe("DataWindow.Syntax")
```

Syntax.Data

Description — The data in the DataWindow object described in parse format (the format required by the DataWindow parser).

Applies to — DataWindow

Used in — Describe

Syntax — "DataWindow.Syntax.Data "

Syntax file
Use this attribute with the Syntax attribute to obtain the description of the DataWindow object and the data. Using this information, you can create a syntax file that represents both the structure and data of a DataWindow at an instance in time. You can then use the syntax file as a DropDownDataWindow containing redefined data at a single location or to mail this as a text object.

Syntax.Modified

Description — Whether the DataWindow syntax has been modified by a function call or user intervention. Calling the Modify, SetSort, or SetFilter function or changing the size of the DataWindow grid automatically sets Syntax.Modified to Yes.

Applies to — DataWindow

Used in — Describe, Modify

Syntax

"DataWindow.Syntax.Modified { = *value* }"

Parameter	Description
value	Whether the DataWindow syntax has been modified. Values are: ♦ Yes — DataWindow syntax has been modified ♦ No — DataWindow has not been modified

Tip
Use this attribute in Modify to set Syntax.Modified to No after you cause a change in the syntax that does not affect the user (such as setting preview on).

Examples

```
setting = dw_1.Describe("DataWindow.Syntax.Modified")
dw_1.Modify("DataWindow.Syntax.Modified=No")
```

Table (for Create)

Description

The section of the DataWindow syntax that specifies information about the DataWindow's database table, including the name of the update table.

Used in

Create, as shown in DataWindow syntax (Appendix B)

Notes

Use this attribute to redefine a DataWindow result set. You can add a column, change the data type of a column, or make other changes to in the table section of your DataWindow involving attributes that are not accessible through normal Modify calls.

Caution
When you use this attribute to redefine the result set, you must redefine the table section in its entirety.

You can call the GetItem and SetItem functions to access columns added using this attribute, but the columns do not display in the DataWindow unless you call Modify("create column(...)") to add them.

❖ To redefine your table section:

1 Export your DataWindow object to a DOS file.

2 Copy *only* the table section into your script.

3 Modify the table section to meet your needs.

4 Put the new table definition into a string variable. Change existing double quotation marks (") in the string to single quotation marks (') and change the tilde quotation marks to tilde tilde single quotation marks (~~').

5 Call Modify. Modifying the table section of your DataWindow causes the DataWindow to be reset.

6 (Optionally) Call Modify to add the column to the DataWindow display.

Table (for TableBlobs)

Description: The name of the database table that contains the blob.

Applies to: TableBlob objects

Used in: Describe, Modify

Syntax: "*tblobname*.Table { = '*tablename* ' }"

Parameter	Description
tblobname	The name of the TableBlob object in the DataWindow.
tablename	(*exp*) A string specifying the name of the table that contains the blob data when you use Modify. *Tablename* can be a quoted DataWindow painter expression.

Examples:

```
setting = dw_1.Describe("blob_1.Table")
dw_1.Modify("blob_1.Table='emp_pictures'")
```

Table.*attribute*

Description: Attributes for the DataWindow's DBMS connection.

Applies to: DataWindow

Used in: Describe, Modify, except where noted

Syntax

"DataWindow.Table.*attribute* { = *value* }"

Parameter	Description
attribute	An attribute for the DataWindow's DBMS connection. Attributes and appropriate values are listed in the table below.
value	The value to be assigned to the attribute when you use Modify.

Attribute for Table	Value
CrosstabData	A string containing a tab-separated list of the expressions used to calculate the values of columns in a crosstab DataWindow.
Filter	(*exp*) A string containing the filter for the DataWindow. Filters are expressions that can evaluate to TRUE or FALSE. The Table.Filter attribute filters the data before it is retrieved. To filter data already in the DataWindow's buffers, use the Filter attribute or the SetFilter and Filter functions. The filter string can be a quoted DataWindow painter expression.
GridColumns	(*Describe only*) The grid columns of a DataWindow.
Procedure	A string that contains the number of the result set returned by the stored procedure to populate the DataWindow object. You can use this attribute only if your DBMS supports stored procedures. Use this attribute to change the stored procedure or to change the data source from a SELECT statement or script to a stored procedure (see the example).
Select	A string containing the SQL SELECT statement that is the data source for the DataWindow. Use this attribute to specify a new SELECT statement or change the data source from a stored procedure or Script to a SELECT statement. Table.Select has several advantages over the SetSQLSelect function: ♦ It is faster. PowerBuilder does not validate the statement until retrieval.

Attribute for Table	Value
	♦ You can change data source for the DataWindow. For example, you can change from a SELECT to a Stored Procedure. ♦ You can use none or any of the arguments defined for the DataWindow object in the SELECT. You cannot use arguments that were not previously defined for the DataWindow object. Describe always tries to return a SQL SELECT statement. If the database is not connected and the attribute's value is a PBSELECT statement, Describe will convert it to a SQL SELECT statement if a SetTransObject function has been executed for the DataWindow object.
Select.Attribute	(*Describe only*) A string containing the SELECT statement for the DataWindow.
Sort	(*exp*) A string containing the sort criteria for the DataWindow, for example, "1A,2D" (column 1 ascending, column 2 descending). The Table.Sort attribute sorts the data before it is retrieved. To sort data already in the DataWindow's buffers, the SetSort and Sort functions. The value for Sort is quoted and can be a DataWindow painter expression.
SQLSelect	The most recently executed SELECT statement. Setting this has no effect. See Select in this table.
UpdateKey-InPlace	Whether the key column can be updated in place or whether the row has to be deleted and reinserted. This value determines the syntax PowerBuilder will generate when a user modifies a key field. ♦ Yes — Use the UPDATE statement when the key is changed so that the key is updated in place ♦ No — Use a DELETE and an INSERT statement when the key is changed ***Caution*** *When there are multiple rows in a DataWindow object and the user switches keys or rows, updating in place may fail due to DBMS duplicate restrictions.*
UpdateTable	A string specifying the name of the database table used to build the Update syntax.

Attribute for Table	Value
UpdateWhere	An integer indicating which columns will be included in the WHERE clause of the Update statement. The value of UpdateWhere can impact performance or cause lost data when more than one user accesses the same tables at the same time. Values are: ♦ 0 — Key columns only (risk of overwriting another user's changes but fast) ♦ 1 — Key columns and all updatable columns (risk of preventing valid updates and slow because SELECT statement is longer) ♦ 2 — Key and modified columns (allows more valid updates than 1 and is faster but not as fast as 0) For more about the effects of this setting, see the discussion of the Specify Update Characteristics dialog in the *User's Guide*.

Examples

```
setting = dw_1.Describe("DataWindow.Table.Sort")

dw_1.Modify("DataWindow.Table.Filter='salary>50000'")

dw_1.Modify &
    (" DataWindow.Table.Procedure= &
    '1 Execute MyOwner MyProcName;1 &
    @NameOfProcArg=:NameOfDWArg,
    @NameOfProcArg=:NameOfDWArg...' ")

sqlvar = 'SELECT ... WHERE ...'
dw_1.Modify("DataWindow.Table.Select='" + sqlvar + "'")
```

TabSequence

Description

The number assigned to the specified object in the DataWindow's tab order. You can also call the SetTabOrder function to change TabSequence.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.TabSequence { = *number* }"

Parameter	Description
columnname	The name of the column whose tab order you want to get or set.
number	A number from 0 to 32000 specifying the position of the column in the tab order when you use Modify. A value of 0 takes the column out of the tab order and makes it read-only.

Examples

```
setting = dw_1.Describe("emp_name.TabSequence")
dw_1.Modify("emp_name.TabSequence = 10")
```

Tag

Description

The tag value of the specified object. The tag value can be any text you see fit to use in your application.

Applies to

Bitmap, Column, Computed Field, Graph, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Tag { = '*string*' }"

Parameter	Description
objectname	The name of an object in the DataWindow.
string	(*exp*) A string specifying the tag for *objectname* when you use Modify. *String* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("blob_1.Tag")
dw_1.Modify("graph_1.Tag = 'Graph of results'")
```

Template

Description

The name of a file that will be used to start the application in OLE.

Applies to

TableBlob objects

Used in

Describe, Modify

Syntax "*tblobname*.Template { = '*string*' }"

Parameter	Description
tblobname	The name of a TableBlob object in the DataWindow.
string	(*exp*) A string whose value is the filename of an application to be the OLE template when you use Modify. *String* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("blob_1.Template")
dw_1.Modify("blob_1.Template='Excel.xls'")
```

Text

Description The text of the specified text object.

Applies to Text objects

Used in Describe, Modify

Syntax "*textname*.Text { = '*string*' }"

Parameter	Description
textname	The name of a text object in the DataWindow.
string	(*exp*) A string specifying the text for *textname* when you use Modify. To specify an accelerator key in the text, include an ampersand before the desired letter. The letter will display underlined. *String* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("text_1.Text")
dw_1.Modify("text_1.Text='Employee &Name'")
```

Timer_Interval

Description The number of milliseconds between the internal timer events. When you use time in a DataWindow, an internal timer event is triggered at the interval specified by Timer_Interval.

Applies to DataWindow

Used in Describe and Modify (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax

Describe and Modify:

"DataWindow.Timer_Interval { = *number* }"

SyntaxFromSQL:

DataWindow(Timer_Interval=*number*)

Parameter	Description
number	An integer specifying the interval between timer events in milliseconds. The default is 0 (which specifies 60,000 or one minute).

Examples

```
setting = dw_1.Describe("DataWindow.Timer_Interval")
dw_1.Modify("DataWindow.Timer_Interval=10000")
```

Title

Description

The title of the graph.

Applies to

Graph objects

Used in

Describe, Modify

Syntax

"*graphname*.Title { = '*titlestring*' }"

Parameter	Description
graphname	The name of the graph object in the DataWindow object for which you want to get or set the title
titlestring	A string specifying the graph's title

Examples

```
setting = dw_1.Describe("gr_1.Title")
dw_1.Modify("gr_1.Title = 'Sales Graph'")
```

Title.DispAttr.*fontattribute*

See DispAttr.*fontattribute*

Trail_Footer

Description

Whether the footer of a nested report is displayed at the end of the report or at the bottom of the page. Trail_Footer only applies to reports in a composite DataWindow. Setting Trail_Footer to No forces objects following the report onto a new page.

Applies to

Report objects

Used in

Describe, Modify

Syntax

"*reportname*.Trail_Footer { = *value* }"

Parameter	Description
reportname	The name of the report object for which you want to get or set Trail_Footer.
value	Whether report's footer trails the last line of the report or appears at the bottom of the page. Values are: ♦ Yes — The footer appears right after the last line of data in the report ♦ No — The footer appears at the bottom of the page, forcing any data following the report onto the following page

Examples

```
setting = dw_1.Describe("rpt_1.Trail_Footer")
dw_1.Modify("rpt_1.Trail_Footer = Yes")
```

Trailer.#.*attribute*

See *Bandname.attribute*

Type

Description

The type of the object (for Describe) or the type of presentation style (for SyntaxFromSQL).

Used in

Describe (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax

Describe:

"*objectname*.Type"

Parameter	Description
objectname	The name of the object for which you want the type. Returned values are: bitmap, column, compute (for computed field), graph, line, ellipse (for oval), rectangle, report, roundrectangle, tableblob, text

SyntaxFromSQL:

Style(Type = *value*)

Parameter	Description
value	A keyword specifying the presentation style for the DataWindow object. Keywords are: ♦ (Default) Tabular ♦ Grid ♦ Form (for the freeform style) ♦ Crosstab ♦ Graph ♦ Group ♦ Label ♦ NUp

Examples

```
setting = dw_1.Describe("emp_name.Type")

SQLCA.SyntaxFromSQL(sqlstring, &
    'Style(... Type=grid ...)', errstring)
```

Units

Description

The unit of measure used to specify measurements in the DataWindow object. You set this in the DataWindow Style dialog when you define the DataWindow object.

Applies to

DataWindow

Used in

Describe (1st syntax), SyntaxFromSQL (2nd syntax)

Syntax

Describe:

"DataWindow.Units"

SyntaxFromSQL:

DataWindow(Units=*value*)

Parameter	Description
value	The type of units for measurements in the DataWindow. Values are: ♦ 0 — PowerBuilder units ♦ 1 — Display pixels ♦ 2 — 1/1000 of a logical inch ♦ 3 — 1/1000 of a logical centimeter

Example

```
setting = dw_1.Describe("DataWindow.Units")
```

Note
PowerBuilder units and display pixels are adjusted for printing.

Update

Description

Whether the specified column is updatable. Each updatable column is included in the SQL statement that the Update function sends to the database. All updatable columns should be in the same database table.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.Update { = *value* }"

Parameter	Description
columnname	The column for which you want to get or set its updatable status.
value	Whether the column is updatable. Values are: ♦ Yes — Include the column in the SQL statement for updating the database ♦ No — Do not include the column in the SQL statement

Examples

```
setting = dw_1.Describe("emp_name.Update")
dw_1.Modify("emp_name.Update=No")
```

Validation

Description

The validation expression for the specified column. Validation expressions are expressions that evaluate to TRUE or FALSE. They provide checking of data that the user enters in the DataWindow.

To set the validation expression, you can also use the SetValidate function. To find out the current validation expression, use the GetValidate function.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.Validation { = '*validationstring* ' }"

Parameter	Description
columnname	The column for which you want to get or set the validation rule.
validationstring	(*exp*) A string containing the rule that will be used to validate data entered in the column. Validation rules are expressions that evaluate to TRUE or FALSE. *Validationstring* is quoted and can be a DataWindow painter expression.

Example

```
setting = dw_1.Describe("emp_status.Validation")
```

ValidationMsg

Description

The message that PowerBuilder displays instead of the default message when an ItemError event occurs in the column.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.ValidationMsg { = '*string*' }"

Parameter	Description
columnname	The column for which you want to get or set the error message displayed when validation fails.
string	(*exp*) A string specifying the error message you want to set when you use Modify. *String* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_salary.ValidationMsg")

dw_1.Modify("emp_salary.ValidationMsg = "  &
    "'Salary must be between 10,000 and 100,000'")
```

Values (for column)

Description

The values in the code table for the column.

Applies to

Column objects

Used in

Describe, Modify

Syntax

"*columnname*.Values { = '*string* ' }"

Parameter	Description
columnname	The column for which you want to specify the contents of the code table.
string	(*exp*) A string containing the code table values for the column. In the string, separate the display values and the actual values with a tab character, and separate multiple pairs of values with a slash using this format: "*displayval*~t*actualval*/*displayval*~t*actualval*/..." For example: "red~t1/white~t2" *String* is quoted and can be a DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_status.Values")

dw_1.Modify("emp_status.Values="  &
   + "'Active~tA/Part Time~tP/Terminated~tT'")
```

Values (for graph)

See *Axis* and *Axis.attribute*

Vertical_Size

Description

The height of the columns in the detail area of the DataWindow object. Vertical_Size is meaningful *only* when Type is Form (meaning the freeform style). When a column reaches the specified height, PowerBuilder starts a new column to the right of the current column. The space between columns is specified in the Vertical_Spread attribute.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Vertical_Size = *value*)

Parameter	Description
value	An integer specifying the height of the columns in the detail area of the DataWindow object area in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring, &
    'Style(... Vertical_Size=1225...)', errstring)
```

Vertical_Spread

Description

The vertical space between columns in the detail area of the DataWindow object. Vertical_Spread is meaningful *only* when Type is Form (meaning the freeform style). The Vertical_Size attribute determines when to start a new column.

Applies to

Style keyword

Used in

SyntaxFromSQL

Syntax

Style(Vertical_Spread = *value*)

Parameter	Description
value	An integer specifying the vertical space between columns in the detail area of the DataWindow object area in the units specified for the DataWindow

Example

```
SQLCA.SyntaxFromSQL(sqlstring,  &
    'Style(... Vertical_Spread=25...)', errstring)
```

VerticalScrollMaximum

Description

The maximum height of the scroll box of the DataWindow's horizontal scrollbar. This value is set by PowerBuilder based on the content of the DataWindow. Use VerticalScrollMaximum with VerticalScrollPosition to synchronize vertical scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe

Syntax

"DataWindow.VerticalScrollMaximum"

Example

```
setting =   &
   dw_1.Describe("DataWindow.VerticalScrollMaximum")
```

VerticalScrollPosition

Description

The position of the scroll box in the vertical scrollbar. Use VerticalScrollMaximum with VerticalScrollPosition to synchronize vertical scrolling in multiple DataWindow objects.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.VerticalScrollPosition { = *scrollvalue* }"

Parameter	Description
scrollvalue	An integer specifying the position of the scroll box in the vertical scrollbar of the DataWindow

Examples

```
spos1 =   &
   dw_1.Describe("DataWindow.VerticalScrollPosition")

smax =  &
   dw_1.Describe("DataWindow.VerticalScrollMaximum")
sscroll = String(Integer(smax)/2)
modstring =  &
   "DataWindow.VerticalScrollPosition=" + sscroll
dw_1.Modify(modstring)
```

Visible

Description

Whether the specified object in the DataWindow is visible.

Applies to

Bitmap, Column, Computed Field, Graph, Line, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Visible { = '*value* ' }"

Parameter	Description
objectname	The name of the object for which you want to get or set the Visible attribute.
value	(*exp*) Whether the specified object is visible. Values are: ♦ 0 — False; the object is not visible ♦ 1 — True; the object is visible *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_status.Visible")

dw_1.Modify("emp_status.Visible=0")

dw_1.Modify("emp_stat.Visible='0~tIf(emp_class=1,0,1)'")
```

Width

Description

The width of the specified object.

Applies to

Bitmap, Column, Computed Field, Graph, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Width { = '*value* ' }"

Parameter	Description
objectname	The name of the object for which you want to get or set the width.
value	(*exp*) The width of the *objectname* in the units specified for the DataWindow. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_name.Width")
dw_1.Modify("emp_name.Width=250")
```

X

Description

The distance of the specified object from the left edge of the DataWindow object.

Applies to

Bitmap, Column, Computed Field, Graph, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.X { = '*value*' }"

Parameter	Description
objectname	The name of the object for which you want to get or set the x coordinate.
value	(*exp*) An integer specifying the x coordinate of the object in the unit of measure specified for the DataWindow object. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_name.X")
dw_1.Modify("emp_name.X=10")
```

X1, X2

Description

The distance of each end of the specified line from the left edge of the line's band.

Applies to

Line objects

Used in

Describe, Modify

Syntax

"*objectname*.X1{ = '*value*' }"

"*objectname*.X2{ = '*value*' }"

Parameter	Description
objectname	The name of the line for which you want to get or set one of the x coordinates.

Parameter	Description
value	(*exp*) An integer specifying the x coordinate of the line in the unit of measure specified for the DataWindow object. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("line_1.X1")

dw_1.Modify("line_1.X1=10")
dw_1.Modify("line_1.X2=1000")
```

Y

Description

The distance of the specified object from the top of the object's band.

Applies to

Bitmap, Column, Computed Field, Graph, Oval, Rectangle, Report, RoundRectangle, TableBlob, Text objects

Used in

Describe, Modify

Syntax

"*objectname*.Y{ = '*value* ' }"

Parameter	Description
objectname	The name of the object for which you want to get or set the y coordinate.
value	(*exp*) An integer specifying the y coordinate of the object in the unit of measure specified for the DataWindow object. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("emp_name.Y")

dw_1.Modify("emp_name.Y=100")
```

Y1, Y2

Description

The distance of each end of the specified line from the top of the line's band.

Applies to

Line objects

Used in

Describe, Modify

Syntax

"*objectname*.Y1{ = '*value* ' }"

"*objectname*.Y2{ = '*value* ' }"

Parameter	Description
objectname	The name of the line for which you want to get or set one of the y coordinates.
value	(*exp*) An integer specifying the y coordinate of the line in the unit of measure specified for the DataWindow object. *Value* can be a quoted DataWindow painter expression.

Examples

```
setting = dw_1.Describe("line_1.Y1")

dw_1.Modify("line_1.Y1=50")
dw_1.Modify("line_1.Y2=50")
```

Zoom

Description

The scaling percentage of the DataWindow object.

Applies to

DataWindow

Used in

Describe, Modify

Syntax

"DataWindow.Zoom { = *value* }"

Parameter	Description
value	An integer specifying the scaling percentage of the DataWindow object. The default is 100%.

Examples

```
setting = dw_1.Describe("DataWindow.Zoom")

dw_1.Modify("DataWindow.Zoom=50")
```

APPENDIX B

DataWindow Syntax Listing

This appendix explains how to create the DataWindow definition syntax and provides sample syntax.

Viewing the actual syntax
To view actual DataWindow syntax, build a DataWindow and then export the definition to a file. You can then print the syntax or view it on screen in a text editor.

Creating syntax

You will use dwSyntaxFromSQL to create the syntax for most dynamic DataWindows. However, to use some of the more advanced dynamic DataWindows features, you will need to write the syntax yourself. For example, you will have to create the syntax yourself to create a group break or to add a column to the DataWindow.

The *User's Guide* describes the statements used to create your own DataWindow syntax. Before you try to build your own syntax, you should examine syntax from one or more of your existing DataWindow objects.

Information about the statements used to create your own DataWindow syntax is also available in online Help. Help also has a syntax listing and information about each of the attributes in the listing.

Comments

To designate text in your DataWindow syntax as a comment, use either of the standard PowerBuilder methods:

- Use double slashes (//) to indicate that the text following the slashes and on the same line is a comment.
- Begin a comment with slash asterisk (/*) and end it with asterisk slash (*/) to indicate that all the text between the delimiters is a comment.

When you use the first method (//), the comment can be all or part of a line but cannot cover multiple lines; the compiler ignores everything following the double slashes on the line.

When you use the second method (/* and */) to designate a comment, the comment can be all or part of a line or multiple lines; the compiler ignores everything between /* and */.

Syntax listing

The order of the syntax can vary. Release must be the first statement and DataWindow should be the next statement.

The order presented here is a logical order. If you change the order, use care; the order can affect the results.

Release 4;
DataWindow
(Color = number constant
Grid.ColumnMove = *yes/no*
Grid.Lines = *integer*
Help.Command = *integer*
Help.File = "*string*"
Help.TypeID = "*string*"
Help.TypeID.*attribute* = "*string*"
Label.Columns = *integer*
Label.Columns.Spacing = *integer*
Label.Ellipse_height = *integer*
Label.Ellipse_width = *integer*
Label.Height = *integer*
Label.Name = "*string*"
Label.Rows = *integer*
Label.Rows.spacing = *integer*
Label.Shape ="*string*"
Label.Sheet = *yes/no*
Label.Topdown = *boolean number*
Label.Width = *integer*
Message.Title = "*string*"
OLE.Client.Class = "*string*"
OLE.Client.Name = "*string*"
Pointer = "*string*" (expression)
Print.Collate = *yes/no*
Print.Color = *integer constant*
Print.Columns = *integer*
Print.Columns.Width = *integer*
Print.Copies = *number constant*
Print.DocumentName = "*string constant*"
Print.Duplex = *number* "
Print.Filename = "*string*"
Print.Margin.Bottom = *number constant*
Print.Margin.Left = *number constant*
Print.Margin.Right = *number constant*
Print.Margin.Top = *number constant*
Print.Orientation = *integer*
Print.Page.Range = "*string*"
Print.Page.RangeInclude = *integer*
Print.Paper.Size = *number*
Print.Paper.Source = *number*
Print.Preview = *yes/no*
Print.Preview.Rulers = *yes/no*
Print.Preview.Zoom = *integer*
Print.Prompt = *yes/no*
Print.Quality = *number*
Print.Scale = *integer*
Processing = *number constant*
ReadOnly = *yes/no*
Row.Resize = *boolean integer*
Rows_Per_Detail = *integer constant*
Selected.Mouse = *yes/no*
Timer_Interval = *number constant*

```
        Units = unit number constant
        Zoom = integer )
    Table
        ( Arguments = (argument entry {,argument
            entry} )
        Column (
            Compute = expression
            DBName = "string expression"
            ID = number constant
            Initial = "string expression"
            Key = yes/no
            Name = name
            Type = column type
            Update = yes/no
            Validation = boolean expression
            ValidationMsg = string expression
            Values = "string constant" )
        Data = "string"
        Filter = boolean expression
        Procedure = procedure
        RETRIEVE = select statement
        Retrieve.AsNeeded = yes/no
        Sort = expression
        Update = "string constant"
        UpdateKeyInPlace = yes/no
        UpdateWhere = "update where")
    Column
        ( Accelerator = accelerator letter
        Alignment = alignment code
        Background.attribute = value expression
        Band = band
        BitmapName = yes/no
        Border = border expression
        CheckBox.LeftText = yes/no
        CheckBox.Off = string constant
        CheckBox.On = string constant
        CheckBox.Other = string constant
        CheckBox.Text = string constant
        Color = number expression
        Criteria.Dialog = yes/no
        Criteria.Override_Edit = yes/no
        Criteria.Required = yes/no
        dddw.AllowEdit = yes/no
        dddw.AutoHScroll = yes/no
        dddw.case = number constant
        dddw.DataColumn = number expression
        dddw.DisplayColumn = number expression
        dddw.HScrollBar = yes/no
        dddw.HSplitScroll = number expression
        dddw.Limit = number constant
        dddw.name = name
        dddw.NilisNull = yes/no
        dddw.PercentWidth = percent
        dddw.Required = yes/no
        dddw.ShowList = yes/no
        dddw.VScrollBar = yes/no
        ddlb.AllowEdit = yes/no
        ddlb.AutoHScroll = yes/no
        ddlb.Limit = number constant
        ddlb.name = name
        ddlb.NilisNull = yes/no
        ddlb.Required = yes/no
        ddlb.ShowList = yes/no
        ddlb.Sorted = yes/no
        ddlb.UseAsBorder = yes/no
        ddlb.VScrollBar = yes/no
        Edit.AutoHScroll = yes/no
        Edit.AutoSelect = yes/no
        Edit.AutoVScroll = yes/no
        Edit.Case = number constant
        Edit.CodeTable = yes/no
        Edit.DisplayOnly = yes/no
        Edit.Format = string expression
        Edit.FocusRectangle = yes/no
        Edit.HScrollBar = yes/no
        Edit.Limit = number constant
```

Edit.Name = *yes/no*
Edit.NilisNull = *yes/no*
Edit.Password = *yes/no*
Edit.Required = *yes/no*
Edit.ValidateCode = *yes/no*
Edit.VScrollBar = *yes/no*
EditMask.FocusRectangle = *yes/no*
EditMask.Name = *name*
EditMask.Mask = *string*
Font.*attribute* = *value expression*
Format = *string expression*
Height = *unit number expression*
Height.AutoSize = *yes/no*
ID = *number constant*
Moveable = *boolean integer*
Name = *name*
Pointer = *string expression*
RadioButtons.Columns = *number constant*
RadioButtons.LeftText = *yes/no*
Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
TabSequence = *number constant*
Tag = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*

Group
(BY = (*column number, column number, ...*)
Header.Color = *long expression*
Header.Height = *unit number constant*
Header.Pointer = *string expression*
Level = *number constant*
NewPage = *yes/no*
ResetPageCount = *yes/no*
Sort = *list*
Trailer.Color = *long expression*
Trailer.Height = *unit number constant*
Trailer.Pointer = *string expression*)

Bitmap
(Band = *band*
Border = *number expression*
Filename = *string expression*
Height = *unit number expression*
Invert = *yes/no*
Moveable = *boolean number expression*
Name = *name*
Pointer = *string expression*
Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Tag = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*)

Compute
(Alignment = *alignment value*
background
Band = *band*
Border = *border value*
Color = *number expression*
Expression = *compute expression*
Font.*attribute* = *value expression*
Format = *string expression*
Height = *unit number expression*
Height.AutoSize = *yes/no*
Name = *name*
Pointer = *string expression*
Resizeable = *boolean integer*
Tag = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*)

Detail
(Height = *number constant*
Color = *number expression*
Height.AutoSize = *yes/no*
Pointer = *number expression*)
Footer
(Height = *number constant*
Color = *number expression*
Pointer = *number expression*)
Graph
(BackColor = *backcolor*
Band = *band*
Border = *border value*
Category = *expression*
Category.Autoscale = *boolean number expression*
Category.Dispattr.*attribute* = *value expression*
Category.DisplayEveryNLabels = *integer expression*
Category.DropLines = *number expression*
Category.Frame = *number expression*
Category.Label = *string expression*
Category.LabelDispattr.*attribute* = *value expression*
Category.MajorDivisions = *number expression*
Category.MajorGridline = *number expression*
Category.MajorTic = *number expression*
Category.MaximumValue = *number expression*
Category.MinimumValue = *number expression*
Category.MinorDivisions = *number expression*
Category.MinorGridline = *number expression*
Category.MinorTic = *number expression*
Category.OriginLine = *number expression*
Category.PrimaryLine = *number expression*
Category.RoundTo = *number expression*
Category.ScaleType = *integer expression*
Category.ScaleValue = *integer expression*
Category.SecondaryLine = *number expression*
Category.ShadeBackEdge = *boolean number expression*
Color = *number expression*
Depth = *number expression*
Elevation = *integer expression*
GraphType = *unit number*
Height = *unit number expression*
Legend = *unit number expression*
Legend.Dispattr.*attribute* = *value expression*
Moveable = *boolean number expression*
Name = *name*
OverlapPercent = *integer expression*
Perspective = *integer expression*
Pointer = *string expression*
Range = *unit number*
Resizeable = *boolean integer*
Rotation = *integer expression*
Series = *expression*
Series.Autoscale = *boolean number expression*
Series.DisplayEveryNLabels = *integer expression*
Series.Dispattr.*attribute* = *value expression*
Series.DropLines = *number expression*
Series.Frame = *number expression*
Series.Label = *string expression*
Series.LabelDispattr.*attribute* = *value expression*
Series.MajorDivisions = *number expression*
Series.MajorGridLine = *number expression*
Series.MajorTic = *number expression*
Series.MaximumValue = *number expression*
Series.MinimumValue = *number expression*
Series.MinorDivisions = *number expression*
Series.MinorGridLine = *number expression*
Series.MinorTic = *number expression*
Series.OriginLine = *number expression*
Series.PrimaryLine = *number expression*
Series.RoundTo = *number expression*
Series.ScaleType = *integer expression*

Series.ScaleValue = *integer expression*
Series.SecondaryLine = *number expression*
Series.ShadeBackEdge = *boolean number expression*
ShadeColor = *number expression*
SizeToDisplay = *boolean number expression*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Spacing = *integer expression*
Tag = *string expression*
Title = *string expression*
Title.Dispattr.*attribute* = *value expression*
Values = *expression*
Values.AutoScale = *boolean number expression*
Values.DisplayEveryNLabels = *integer expression*
Values.Dispattr.*attribute* = *value expression*
Values.DropLines = *number expression*
Values.Frame = *number expression*
Values.Label = *string expression*
Values.LabelDispattr.*attribute* = *value expression*
Values.MajorDivisions = *number expression*
Values.MajorGridLine = *number expression*
Values.MajorTic = *number expression*
Values.MaximumValue = *number expression*
Values.MinimumValue = *number expression*
Values.MinorDivisions = *number expression*
Values.MinorGridLine = *number expression*
Values.MinorTic = *number expression*
Values.OriginLine = *number expression*
Values.PrimaryLine = *number expression*
Values.RoundTo = *number expression*
Values.ScaleType = *integer expression*
Values.ScaleValue = *integer expression*
Values.SecondaryLine = *number expression*
Values.ShadeBackEdge = *boolean number expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*)

Header
(Color = *number expression*
Height = *number constant*
Pointer = *number expression*)

Line
(Background = *background*
Band = *band*
Moveable = *boolean number*
Name = *name*
Pen.*attribute* = *value expression*
Pointer = *string expression*
Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Tag = *string expression*
Visible = *boolean number expression*
X1 = *unit number expression*
X2 = *unit number expression*
Y1 = *unit number expression*
Y2 = *unit number expression*)

Oval
(Background.*attribute* = *value expression*
Band = *band*
Brush.*attribute* = *value expression*
Height = *unit number expression*
Moveable = *boolean number expression*
Name = *name*
Pen.*attribute* = *value expression*
Pointer = *string expression*
Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Tag = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*

X = *unit number expression*

Y = *unit number expression*)

Note: Oval and Ellipse are synonyms in dw syntax.

Rectangle

(Background.*attribute* = *value expression*

Band = *band*

Brush.*attribute* = *value expression*

Height = *unit number expression*

Moveable = *boolean number expression*

Name = *name*

Pen.*attribute* = *value expression*

Pointer = *string expression*

Resizeable = *boolean integer*

SlideLeft = *boolean expression*

SlideUp = *slide value expression*

Tag = *string expression*

Visible = *boolean number expression*

Width = *unit number expression*

X = *unit number expression*

Y = *unit number expression*)

RoundRectangle

(Background.*attribute* = *value expression*

Band = *band*

Brush.*attribute* = *value expression*

EllipseHeight = *unit number expression*

EllipseWidth = *unit number expression*

Height = *unit number expression*

Moveable = *boolean number expression*

Name = *name*

Pen.*attribute* = *value expression*

Pointer = *string expression*

Resizeable = *boolean integer*

SlideLeft = *boolean expression*

SlideUp = *slide value expression*

Tag = *string expression*

Visible = *boolean number expression*

Width = *unit number expression*

X = *unit number expression*

Y = *unit number expression*)

Report

(Band = *band*

Border = *number expression*

Criteria = *number expression*

Dataobject = *string*

Height = *unit number expression*

Height.AutoSize = *yes/no*

Moveable = *boolean number expression*

Name = *name*

Nest_Arguments = *string*

Nested = *yes/no*

New_Page = *value*

Pointer = *string expression*

Resizeable = *boolean integer*

SlideLeft = *boolean expression*

SlideUp = *slide value expression*

Tag = *string expression*

Trail_Footer = *yes/no*

Visible = *boolean number expression*

Width = *unit number expression*

X = *unit number expression*

Y = *unit number expression*)

Summary

(Height = *unit number expression*

Color = *number expression*)

TableBlob

(Band = *band*

Border = *number expression*

ClientName = *string expression*

Height = *unit number expression*

ID = *column number*

KeyClause = *string expression*

Moveable = *boolean number expression*

Name = *name*

OLEClass = *string expression*

Pointer = *string expression*

Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Table = *string expression*
Tag = *string expression*
Template = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*)

Text
(Alignment = *alignment expression*
Background.*attribute* = *value expression*
Band = *band*
Border = *number expression*
Color = *number expression*
Font.*attribute* = *value expression*
Height = *unit number expression*
Height.AutoSize = *yes/no*
Moveable = *boolean number expression*
Name = *name*
Pointer = *string expression*
Resizeable = *boolean integer*
SlideLeft = *boolean expression*
SlideUp = *slide value expression*
Tag = *string expression*
Text = *string expression*
Visible = *boolean number expression*
Width = *unit number expression*
X = *unit number expression*
Y = *unit number expression*)

Sparse
(names = *string expression*)

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z